Three.js Manual

Complete guide to the Three.js JavaScript 3D library

Source: https://threejs.org/manual/

Generated: 2026-08-10 v0.8.10-2-gd493cab

How to Read This Book

This reference book was generated from the publicly accessible documentation at `https://threejs.org/manual/` on 2026-08-10. It is a structured, self-contained snapshot suitable for offline reference and AI ingestion.

Table of Contents

Loading 3D Models

How to load 3D models into your three.js scenes, including OBJ and GLTF formats.

Advanced Rendering

Advanced rendering techniques: post-processing, render targets, custom BufferGeometry, and Shadertoy integration.

Physics, Games & Interaction

Building interactive experiences with picking, physics, games, voxel geometry, and HTML-to-3D alignment.

- Voxel(Minecraft Like) Geometry
- Aligning HTML Elements to 3D

WebGPU

Using the WebGPURenderer and post-processing effects with WebGPU.

- WebGPURenderer
- Post-Processing with WebGPURenderer

Virtual Reality

Creating VR content with three.js, including look-to-select and 3DOF point-to-select interactions.

- How to create VR content
- VR
- VR - Look to Select
- VR - 3DOF Point to Select

Optimization & Performance

Techniques for optimizing three.js applications: rendering on demand, general tips, managing many objects, animated objects, and OffscreenCanvas.

- Rendering on Demand
- Tips
- Optimize Lots of Objects
- Optimize Lots of Objects Animated
- OffscreenCanvas

UX & UI Integration

Responsive design, multiple canvases and scenes, and billboard techniques.

- Responsive Design
- Multiple Canvases Multiple Scenes
- Billboards

Maintenance & Debugging

Object disposal, updating things, cleanup routines, and debugging GLSL shaders.

- How to dispose of Objects
- How to update Things
- Cleanup
- Debugging - GLSL

Compatibility & FAQ

WebGL compatibility checks and answers to frequently asked questions.

- WebGL Compatibility Check
- FAQ

Reference

Reference materials: libraries and plugins, uniform types, useful links, and the material feature table.

- Libraries and Plugins
- Uniform Types
- Useful Links
- Material Feature Table

Introduction

three.js manual

OVERVIEW

Source: https://threejs.org/manual/

The main index page for the three.js manual, listing all available guides, tutorials, references, and solutions organized by topic category including Getting Started, Fundamentals, Tips, Optimization, Solutions, WebGPU, and WebXR.

Getting Started

- Installation
- Creating a Scene
- Creating Text
- Drawing Lines

- FAQ
- Libraries and Plugins
- Loading 3D Models
- Uniform Types
- Useful Links
- WebGL Compatibility Check

Next Steps

- Animation System
- Color Management
- How to create VR content
- How to dispose of Objects
- How to update Things
- How to use Post Processing
- Matrix Transformations

Basics

- Fundamentals
- Responsive Design
- Prerequisites
- Setup

Fundamentals

- Primitives
- Scenegraph
- Materials
- Textures
- Lights
- Cameras
- Shadows
- Fog
- Render Targets
- Custom BufferGeometry
- Physics

Tips

- Rendering On Demand
- Debugging JavaScript
- Debugging GLSL
- Taking a screenshot
- Prevent the Canvas Being Cleared
- Get Keyboard Input From a Canvas
- Make the Canvas Transparent
- Use three.js as Background in HTML

Optimization

- Optimizing Lots of Objects
- Optimizing Lots of Objects Animated
- Using OffscreenCanvas in a Web Worker

Solutions

- Load an .OBJ file
- Load a .GLTF file
- Add a Background or Skybox
- How to Draw Transparent Objects
- Multiple Canvases, Multiple Scenes
- Picking Objects with the mouse
- Post Processing
- Using Shadertoy shaders
- Aligning HTML Elements to 3D
- Using Indexed Textures for Picking and Color
- Using A Canvas for Dynamic Textures
- Billboards and Facades
- Freeing Resources
- Making Voxel Geometry (Minecraft)
- Start making a Game

WebGPU

- WebGPURenderer

- Post-Processing

WebXR

- VR - Basics
- VR - Look To Select
- VR - Point To Select

Reference

- Material Table

three.js docs

OVERVIEW

Source: https://threejs.org/docs

The main documentation index for three.js, a JavaScript 3D library. This page provides a categorized listing of all classes, modules, and functions available in the Core library, Addons, and the Three Shading Language (TSL).

Core

The Core library provides the fundamental classes and modules for three.js, organized into the following categories: Animation, Audio, Cameras, Core, Extras, Geometries, Helpers, Lights, Loaders, Materials, Math, Nodes, Objects, Renderers, Scenes, and Textures.

Core / Animation

AnimationAction, AnimationClip, AnimationMixer, AnimationObjectGroup, AnimationUtils, BooleanKeyframeTrack, ColorKeyframeTrack, KeyframeTrack, NumberKeyframeTrack, PropertyBinding, PropertyMixer, QuaternionKeyframeTrack, StringKeyframeTrack, VectorKeyframeTrack

Core / Audio

Audio, AudioAnalyser, AudioContext, AudioListener, PositionalAudio

Core / Cameras

ArrayCamera, Camera, CubeCamera, OrthographicCamera, PerspectiveCamera, StereoCamera

Core / Core

BufferAttribute, BufferGeometry, Clock, EventDispatcher, Float16BufferAttribute, Float32BufferAttribute, GLBufferAttribute, InstancedBufferAttribute, InstancedBufferGeometry, InstancedInterleavedBuffer, Int16BufferAttribute, Int32BufferAttribute, Int8BufferAttribute,

InterleavedBuffer, InterleavedBufferAttribute, Layers, Object3D, Raycaster, RenderTarget, RenderTarget3D, Timer, Uint16BufferAttribute, Uint32BufferAttribute, Uint8BufferAttribute, Uint8ClampedBufferAttribute, Uniform, UniformsGroup

Core / Extras

ArcCurve, CatmullRomCurve3, Controls, CubicBezierCurve, CubicBezierCurve3, Curve, CurvePath, DataUtils, Earcut, EllipseCurve, ImageUtils, LineCurve, LineCurve3, PMREMGenerator, Path, QuadraticBezierCurve, QuadraticBezierCurve3, Shape, ShapePath, ShapeUtils, SplineCurve, TextureUtils, Interpolations

Core / Geometries

BoxGeometry, CapsuleGeometry, CircleGeometry, ConeGeometry, CylinderGeometry, DodecahedronGeometry, EdgesGeometry, ExtrudeGeometry, IcosahedronGeometry, LatheGeometry, OctahedronGeometry, PlaneGeometry, PolyhedronGeometry, RingGeometry, ShapeGeometry, SphereGeometry, TetrahedronGeometry, TorusGeometry, TorusKnotGeometry, TubeGeometry, WireframeGeometry

Core / Helpers

ArrowHelper, AxesHelper, Box3Helper, BoxHelper, CameraHelper, DirectionalLightHelper, GridHelper, HemisphereLightHelper, PlaneHelper, PointLightHelper, PolarGridHelper, SkeletonHelper, SpotLightHelper

Core / Lights

AmbientLight, DirectionalLight, DirectionalLightShadow, HemisphereLight, IESSpotLight, Light, LightProbe, LightShadow, PointLight, PointLightShadow, ProjectorLight, RectAreaLight, SpotLight, SpotLightShadow

Core / Loaders

AnimationLoader, AudioLoader, BufferGeometryLoader, Cache, CompressedTextureLoader, CubeTextureLoader, DataTextureLoader, FileLoader, ImageBitmapLoader, ImageLoader, Loader, LoaderUtils, LoadingManager, MaterialLoader, NodeLoader, NodeMaterialLoader, NodeObjectLoader, ObjectLoader, TextureLoader

Core / Materials

Line2NodeMaterial, LineBasicMaterial, LineBasicNodeMaterial, LineDashedMaterial, LineDashedNodeMaterial, Material, MeshBasicMaterial, MeshBasicNodeMaterial, MeshDepthMaterial, MeshDistanceMaterial, MeshLambertMaterial, MeshLambertNodeMaterial, MeshMatcapMaterial, MeshMatcapNodeMaterial, MeshNormalMaterial, MeshNormalNodeMaterial, MeshPhongMaterial, MeshPhongNodeMaterial, MeshPhysicalMaterial, MeshPhysicalNodeMaterial, MeshSSSNodeMaterial, MeshStandardMaterial, MeshStandardNodeMaterial, MeshToonMaterial, MeshToonNodeMaterial, NodeMaterial,

NodeMaterialObserver, PointsMaterial, PointsNodeMaterial, RawShaderMaterial, SSSLightingModel, ShaderMaterial, ShadowMaterial, ShadowNodeMaterial, SpriteMaterial, SpriteNodeMaterial, VolumeNodeMaterial

Core / Math

BezierInterpolant, Box2, Box3, Color, CubicInterpolant, Cylindrical, DiscreteInterpolant, Euler, Frustum, FrustumArray, Interpolant, Line3, LinearInterpolant, MathUtils, Matrix2, Matrix3, Matrix4, Plane, Quaternion, QuaternionLinearInterpolant, Ray, Sphere, Spherical, SphericalHarmonics3, Triangle, Vector2, Vector3, Vector4

Core / Nodes

AONode, AmbientLightNode, AnalyticLightNode, ArrayElementNode, ArrayNode, AssignNode, AtomicFunctionNode, AttributeNode, BarrierNode, BasicEnvironmentNode, BasicLightMapNode, BasicLightingModel, BitcastNode, BitcountNode, BufferAttributeNode, BufferNode, BuiltinNode, BumpMapNode, BypassNode, ClippingNode, CodeNode, ColorSpaceNode, ComputeBuiltinNode, ComputeNode, ConditionalNode, ConstNode, ContextNode, ConvertNode, CubeMapNode, CubeTextureNode, DirectionalLightNode, EnvironmentNode, EventNode, ExpressionNode, FlipNode, FrontFacingNode, FunctionCallNode, FunctionNode, FunctionOverloadingNode, GLSLNodeFunction, GLSLNodeParser, HemisphereLightNode, IESSpotLightNode, IndexNode, InputNode, InspectorNode, IrradianceNode, IsolateNode, JoinNode, LightProbeNode, LightingContextNode, LightingModel, LightingNode, LightsNode, LoopNode, MRTNode, MaterialNode, MaterialReferenceNode, MathNode, MaxMipLevelNode, MemberNode, ModelNode, Node, NodeAttribute, NodeBuilder, NodeCache, NodeCode, NodeError, NodeFrame, NodeFunction, NodeFunctionInput, NodeParser, NodeUniform, NodeVar, NodeVarying, NormalMapNode, Object3DNode, OperatorNode, OutputStructNode, OverrideContextNode, PMREMNode, PackFloatNode, ParameterNode, PassMultipleTextureNode, PassNode, PassTextureNode, PhongLightingModel, PhysicalLightingModel, PointLightNode, PointShadowNode, PointUVNode, ProjectorLightNode, PropertyNode, RTTNode, RangeNode, RectAreaLightNode, ReferenceBaseNode, ReferenceElementNode, ReferenceNode, ReflectorNode, RenderOutputNode, RendererReferenceNode, RotateNode, SampleNode, ScreenNode, SetNode, ShadowBaseNode, ShadowMaskModel, ShadowNode, SplitNode, SpotLightNode, StackNode, StackTrace, StorageArrayElementNode, StorageBufferNode, StorageTexture3DNode, StorageTextureNode, StructNode, StructTypeNode, SubBuildNode, SubgroupFunctionNode, TempNode, Texture3DNode, TextureNode, TextureSizeNode, ToneMappingNode, ToonLightingModel, ToonOutlinePassNode, UniformArrayElementNode, UniformArrayNode, UniformGroupNode, UniformNode, UnpackFloatNode, UserDataNode, VarNode, VaryingNode, VelocityNode, VertexColorNode, ViewportDepthNode, ViewportDepthTextureNode, ViewportSharedTextureNode, ViewportTextureNode, VolumetricLightingModel, WorkgroupInfoElementNode, WorkgroupInfoNode

Core / Objects

BatchedMesh, Bone, ClippingGroup, Group, InstancedMesh, LOD, Line, LineLoop, LineSegments, Mesh, Points, Skeleton, SkinnedMesh, Sprite

Core / Renderers

Backend, BlendMode, BundleGroup, CanvasTarget, CubeRenderTarget, GLSLNodeBuilder, IndirectStorageBufferAttribute, Info, InspectorBase, PostProcessing, QuadMesh, ReadbackBuffer, RenderPipeline, Renderer, StandardNodeLibrary, Storage3DTexture, StorageArrayTexture, StorageBufferAttribute, StorageInstancedBufferAttribute, StorageTexture, TimestampQueryPool, WGSLNodeBuilder, WGSLNodeFunction, WGSLNodeParser, WebGL3DRenderTarget, WebGLArrayRenderTarget, WebGLCubeRenderTarget, WebGLRenderTarget, WebGLRenderer, WebGLTimestampQueryPool, WebGPURenderer, WebGPUTimestampQueryPool, WebXRDepthSensing, WebXRManager, XRManager, UniformsUtils

Core / Scenes

Fog, FogExp2, Scene

Core / Textures

CanvasTexture, CompressedArrayTexture, CompressedCubeTexture, CompressedTexture, CubeDepthTexture, CubeTexture, Data3DTexture, DataArrayTexture, DataTexture, DepthTexture, ExternalTexture, FramebufferTexture, HTMLTexture, Source, Texture, VideoFrameTexture, VideoTexture

Addons

The Addons section provides additional classes and modules extending three.js functionality, organized into the following categories: Animation, Capabilities, Controls, Csm, Curves, Effects, Environments, Exporters, Generators, Geometries, Gpgpu, Helpers, Inspector, Interaction, Interactive, Lighting, Lights, Lines, Loaders, Materials, Math, Misc, Modifiers, Objects, Physics, Postprocessing, Renderers, Shaders, TSL, Textures, Transpiler, Utils, and Webxr.

Addons / Animation

AnimationClipCreator, CCDIKHelper, CCDIKSolver

Addons / Capabilities

WebGL, WebGPU

Addons / Controls

ArcballControls, DragControls, FirstPersonControls, FlyControls, MapControls, OrbitControls, PointerLockControls, TrackballControls, TransformControls

Addons / Csm

CSM, CSMFrustum, CSMHelper, CSMShadowNode, CSMShader

Addons / Curves

CinquefoilKnot, DecoratedTorusKnot4a, DecoratedTorusKnot4b, DecoratedTorusKnot5a, DecoratedTorusKnot5c, FigureEightPolynomialKnot, GrannyKnot, HeartCurve, HelixCurve, KnotCurve, NURBSCurve, NURBSSurface, NURBSVolume, TorusKnot, TrefoilKnot, TrefoilPolynomialKnot, VivianiCurve, NURBSUtils

Addons / Effects

AnaglyphEffect, AsciiEffect, OutlineEffect, ParallaxBarrierEffect, StereoEffect

Addons / Environments

ColorEnvironment, DebugEnvironment, RoomEnvironment

Addons / Exporters

DRACOExporter, EXRExporter, GLTFExporter, KTX2Exporter, OBJExporter, PLYExporter, STLExporter, USDZExporter

Addons / Generators

CityGenerator, FaceFrame, ForestGenerator, SidewalkGenerator, SkyscraperGenerator, TerrainGenerator, TreeGenerator

Addons / Geometries

BoxLineGeometry, ConvexGeometry, DecalGeometry, LoftGeometry, ParametricGeometry, RoundedBoxGeometry, TeapotGeometry, TextGeometry, ParametricFunctions

Addons / Gpgpu

BitonicSort

Addons / Helpers

AnimationPathHelper, LightProbeGridHelper, LightProbeHelper, OctreeHelper, PositionalAudioHelper, RapierHelper, RectAreaLightHelper, TextureHelper, VertexNormalsHelper, VertexTangentsHelper, ViewHelper

Addons / Inspector

Tab

Addons / Interaction

InteractionManager

Addons / Interactive

HTMLMesh, InteractiveGroup, SelectionBox, SelectionHelper

Addons / Lighting

ClusteredLighting, DynamicLighting, LightProbeGrid

Addons / Lights

LightProbeGenerator, RectAreaLightTexturesLib, RectAreaLightUniformsLib

Addons / Lines

Line2, LineGeometry, LineMaterial, LineSegments2, LineSegmentsGeometry, Wireframe, WireframeGeometry2

Addons / Loaders

AMFLoader, BVHLoader, ColladaComposer, ColladaLoader, ColladaParser, DDSLoader, DRACOLoader, EXRLoader, FBXLoader, Font, FontLoader, GCodeLoader, GLTFLoader, HDRCubeTextureLoader, HDRLoader, IESLoader, KMZLoader, KTX2Loader, KTXLoader, LDrawLoader, LUT3dlLoader, LUTCubeLoader, LUTImageLoader, LWOLoader, LottieLoader, MD2Loader, MDDLoader, MTLLoader, MaterialXLoader, NRRDLoader, OBJLoader, PCDLoader, PDBLoader, PLYLoader, PVRLoader, Rhino3dmLoader, STLLoader, SVGLoader, TDSLoader, TGALoader, TIFFLoader, TTFLoader, ThreeMFLoader, USDComposer, USDLoader, UltraHDRLoader, VOXLoader, VRMLLoader, VTKLoader, XYZLoader

Addons / Materials

LDrawConditionalLineMaterial, WoodNodeMaterial

Addons / Math

Capsule, ColorConverter, ConvexHull, ImprovedNoise, Lut, MeshSurfaceSampler, OBB, Octree, SimplexNoise, ColorSpaces

Addons / Misc

ConvexObjectBreaker, GPUComputationRenderer, Gyroscope, MD2Character, MD2CharacterComplex, MorphAnimMesh, MorphBlendMesh, ProgressiveLightMap, RollerCoasterGeometry, RollerCoasterLiftersGeometry, RollerCoasterShadowGeometry, SkyGeometry, TileCreasedNormalsPlugin, TreesGeometry, TubePainter, Volume, VolumeSlice

Addons / Modifiers

EdgeSplitModifier, Flow, InstancedFlow, SimplifyModifier, TessellateModifier

Addons / Objects

GroundedSkybox, Lensflare, LensflareElement, LensflareMesh, MarchingCubes, Reflector, ReflectorForSSRPass, Refractor, ShadowMesh, Sky, SkyMesh, Water, WaterMesh

Addons / Physics

AmmoPhysics, JoltPhysics, RapierPhysics

Addons / Postprocessing

AfterimagePass, BloomPass, BokehPass, ClearMaskPass, ClearPass, CubeTexturePass, DotScreenPass, EffectComposer, FXAAPass, FilmPass, FullScreenQuad, GTAOPass, GlitchPass, HalftonePass, LUTPass, MaskPass, OutlinePass, OutputPass, Pass, RenderPass, RenderPixelatedPass, RenderTransitionPass, SAOPass, SMAAPass, SSAARenderPass, SSAOPass, SSRPass, SavePass, ShaderPass, TAARenderPass, TexturePass, UnrealBloomPass

Addons / Renderers

CSS2DObject, CSS2DRenderer, CSS3DObject, CSS3DRenderer, CSS3DSprite, Projector, SVGObject, SVGRenderer

Addons / Shaders

ACESFilmicToneMappingShader, AfterimageShader, BasicShader, BleachBypassShader, BlendShader, BokehShader, BokehShader2, BrightnessContrastShader, ColorCorrectionShader, ColorifyShader, ConvolutionShader, CopyShader, DOFMipMapShader, DepthLimitedBlurShader, DigitalGlitch, DotScreenShader, ExposureShader, FXAAShader, FilmShader, FocusShader, FreiChenShader, GTAOShader, GammaCorrectionShader, HalftoneShader, HorizontalBlurShader, HorizontalTiltShiftShader, HueSaturationShader, KaleidoShader, LuminosityHighPassShader, LuminosityShader, MirrorShader, NormalMapShader, OutputShader, PoissonDenoiseShader, RGBShiftShader, SAOShader, SMAAShader, SSAOShader, SSRShader, SepiaShader, SobelOperatorShader, SubsurfaceScatteringShader, TriangleBlurShader, UnpackDepthRGBAShader, VelocityShader, VerticalBlurShader, VerticalTiltShiftShader, VignetteShader, VolumeShader, WaterRefractionShader

Addons / TSL

AfterImageNode, AmbientLightDataNode, AnaglyphPassNode, BilateralBlurNode, BloomNode, ChromaticAberrationNode, ClusteredLightsNode, DenoiseNode, DepthOfFieldNode, DirectionalLightDataNode, DotScreenNode, DynamicLightsNode, EnvMapCDFGenerator, FSR1Node, FXAANode, FilmNode, GTAONode, GaussianBlurNode, GodraysNode, HemisphereLightDataNode, ImportanceSampledEnvironment, LensflareNode, Lut3DNode, OutlineNode, ParallaxBarrierPassNode, PixelationNode, PixelationPassNode, PointLightDataNode, RGBShiftNode, RecurrentDenoiseNode, RetroPassNode, SMAANode, SSAAPassNode, SSGINode, SSRNode, SSSNode, SharpenNode, SobelOperatorNode, SpotLightDataNode, StereoCompositePassNode, StereoPassNode, TAAUNode, TRAANode,

TemporalReprojectNode, TileShadowNode, TileShadowNodeHelper, TransitionNode, WebGLNodesHandler, Bayer, GroundedSkybox, Raymarching

Addons / Textures

FlakesTexture

Addons / Transpiler

Transpiler

Addons / Utils

LDrawUtils, SceneOptimizer, ShadowMapViewer, WorkerPool, BufferGeometryUtils, CameraUtils, ColorUtils, GeometryCompressionUtils, GeometryUtils, SceneUtils, SkeletonUtils, SortUtils, UVsDebug, WebGLTextureUtils, WebGPUTextureUtils

Addons / Webxr

ARButton, OculusHandModel, OculusHandPointerModel, VRButton, XRButton, XRControllerModel, XRControllerModelFactory, XREstimatedLight, XRHandMeshModel, XRHandModel, XRHandModelFactory, XRHandPrimitiveModel, XRPlanes, Text2D

TSL

The Three Shading Language (TSL) is a JavaScript-based shading language for three.js. It provides functions for building shader graphs and manipulating nodes in a programmatic way. TSL entries include control flow functions, math operations, lighting models, texture sampling, geometry access, and screen-space effects. Key entries include: Break, Const, Continue, Discard, EPSILON, HALF_PI, INFINITY, If, Loop, PI, PI2, Return, Switch, TBNViewMatrix, TWO_PI, Var, VarIntent, abs, acesFilmicToneMapping, acos, acosh, add, afterImage, agxToneMapping, all, alphaLine, alphaT, ambientOcclusion, anaglyphPass, and, anisotropy, anisotropyB, anisotropyT, any, ao, append, applyVarianceClipping, array, asin, asinh, assign, atan, atanh, atomicAdd, atomicAnd, atomicFunc, atomicLoad, atomicMax, atomicMin, atomicNode, atomicOr, atomicStore, atomicSub, atomicXor, attenuationColor, attenuationDistance, attribute, attributeArray, backgroundBlurriness, backgroundIntensity, backgroundRotation, barrelMask, barrelUV, barrier, batch, beautyTexelFromScreen, bentNormalView, bilateralBlur, billboarding, bitAnd, bitNot, bitOr, bitXor, bitangentGeometry, bitangentLocal, bitangentView, bitangentViewFrame, bitangentWorld, bitcast, bleach, blendBurn, blendColor, blendDodge, blendOverlay, blendScreen, bloom, boxBlur, buffer, bufferAttribute, builtin, builtinAOContext, builtinShadowContext, bumpMap, bypass, cache, cameraFar, cameraIndex, cameraNear, cameraNormalMatrix, cameraPosition, cameraProjectionMatrix, cameraProjectionMatrixInverse, cameraViewMatrix, cameraViewport, cameraWorldMatrix, cbrt, cdl, ceil, checker, chromaticAberration, cineonToneMapping, circle, clamp, clearcoat, clearcoatNormalView, clearcoatRoughness, clipSpace, clipToAABB, clipping, clippingAlpha, clusteredLights, code, collectNeighborhood, colorBleeding, colorSpaceToWorking, colorToDirection, compute,

computeBuiltin, computeFrustumSize, computeHitDistFactor, computeKernel, computeSkinning, context, convertColorSpace, convertToTexture, cos, cosh, countLeadingZeros, countOneBits, countTrailingZeros, createVar, cross, cubeMapNode, cubeTexture, cubeTextureBase, curlNoise, dFdx, dFdy, dashSize, debug, decrement, decrementBefore, degrees, deltaTime, denoise, densityFogFactor, depth, depthBase, depthPass, determinant, difference, diffuseColor, diffuseColorDistance, diffuseContribution, directionToColor, directionToFaceDirection, dispersion, distance, div, dof, dot, dotScreen, drawIndex, dynamicBufferAttribute, dynamicLights, emissive, equal, equirectDirection, equirectUV, exp, exp2, exponentialHeightFogFactor, expression, faceDirection, faceForward, film, floatBitsToInt, floatBitsToUint, floor, fog, fract, frameGroup, frameId, frontFacing, fsr1, fwidth, fxaa, gain, gapSize, gaussianBlur, getNormalFromDepth, getParallaxCorrectNormal, getScreenPosition, getShadowMaterial, getShadowRenderObjectFunction, getSpecularDominantDirection, getTemporalVarianceFactor, getViewPosition, globalId, glsl, godrays, grayscale, greaterThan, greaterThanEqual, hardwareClipping, hash, hashBlur, highpModelNormalViewMatrix, highpModelViewMatrix, hue, increment, incrementBefore, inspector, instance, instanceIndex, instancedArray, instancedBufferAttribute, instancedDynamicBufferAttribute, instancedMesh, intBitsToFloat, interleavedGradientNoise, inverse, inverseSqrt, invocationLocalIndex, invocationSubgroupIndex, ior, iridescence, iridescenceIOR, iridescenceThickness, isolate, js, karisTemporalBlend, label, length, lengthSq, lensflare, lessThan, lessThanEqual, lightPosition, lightProjectionUV, lightShadowMatrix, lightTargetDirection, lightTargetPosition, lightViewPosition, lights, linearDepth, linearToneMapping, lobeNormalFalloff, lobeNormalWeight

Getting Started

Prerequisites

GUIDE

Source: https://threejs.org/manual/en/prerequisites.html

This page outlines the JavaScript and web development knowledge assumed by the three.js articles, including ES6 modules, CSS selectors, closures, ES5/ES6/ES7 features, and recommended tooling like Visual Studio Code with ESLint.

Prerequisites

These articles are meant to help you learn how to use three.js. They assume you know how to program in JavaScript. They assume you know what the DOM is, how to write HTML as well as create DOM elements in JavaScript. They assume you know how to use es6 modules via import and via `<script type="module">` tags. They assume you know how to use import maps. They assume you know some CSS and that you know what CSS selectors are. They also assume you know ES5, ES6 and maybe some ES7. They assume you know that the browser runs JavaScript only via events and callbacks. They assume you know what a closure is.

Here's some brief refreshers and notes

es6 modules

es6 modules can be loaded via the `import` keyword in a script or inline via a `<script type="module">` tag. Here's an example

See more details at the bottom of this article.

```html
<script type="importmap">
{
  "imports": {
    "three": "./path/to/three.module.js",
    "three/addons/": "./different/path/to/examples/jsm/"
  }
}
</script>

<script type="module">
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';

...

</script>
```

`document.querySelector` and `document.querySelectorAll`

You can use `document.querySelector` to select the first element that matches a CSS selector. `document.querySelectorAll` returns all elements that match a CSS selector.

You don't need `onload`

Lots of 20yr old pages use HTML like

That style is deprecated. Put your scripts at the bottom of the page.

or use the `defer` property.

```html
<body onload="somefunction()">
```

```html
<html>
  <head>
    ...
  </head>
  <body>
     ...
  </body>
  <script>
    // inline javascript
  </script>
</html>
```

Know how closures work

In the code above the function `a` creates a new function every time it's called. That function *closes* over the variable `foo`. Here's more info.

```javascript
function a(v) {
  const foo = v;
  return function() {
     return foo;
  };
}

const f = a(123);
const g = a(456);
console.log(f());  // prints 123
console.log(g());  // prints 456
```

Understand how `this` works

`this` is not magic. It's effectively a variable that is automatically passed to functions just like an argument is passed to function. The simple explanation is when you call a function directly like

`this` will be `null` (when in strict mode or in a module) where as when you call a function via the dot operator `.` like this

`this` will be set to `someobject`.

The parts where people get confused is with callbacks.

doesn't work as someone inexperienced might expect because when `loader.load` calls the callback it's not calling it with the dot `.` operator so by default `this` will be null (unless the loader explicitly sets it to something). If you want `this` to be `someobject` when the callback happens you need to tell JavaScript that by binding it to the function.

this article might help explain `this`.

```javascript
somefunction(a, b, c);
```

```javascript
someobject.somefunction(a, b, c);
```

```javascript
const callback = someobject.somefunction;
loader.load(callback);
```

```javascript
const callback = someobject.somefunction.bind(someobject);
loader.load(callback);
```

ES5/ES6/ES7 stuff

`var` is deprecated. Use `const` and/or `let`

There is no reason to use `var` **EVER** and at this point it's considered bad practice to use it at all. Use `const` if the variable will never be reassigned which is most of the time. Use `let` in those cases where the value changes. This will help avoid tons of bugs.

Use `for(elem of collection)` never `for(elem in collection)`

`for of` is new, `for in` is old. `for in` had issues that are solved by `for of`

As one example you can iterate over all the key/value pairs of an object with

```javascript
for (const [key, value] of Object.entries(someObject)) {
  console.log(key, value);
}
```

Use `forEach`, `map`, and `filter` where useful

Arrays added the functions `forEach`, `map`, and `filter` and are used fairly extensively in modern JavaScript.

Use destructuring

Assume an object `const dims = {width: 300, height: 150}`

old code

new code

Destructuring works with arrays too. Assume an array `const position = [5, 6, 7, 1]`;

old code

new code

Destructuring also works in function arguments

```javascript
const width = dims.width;
const height = dims.height;
```

```javascript
const {width, height} = dims;
```

javascript

```
const y = position[1];
const z = position[2];
```

javascript

```
const [, y, z] = position;
```

javascript

```
const dims = {width: 300, height: 150};
const vector = [3, 4];

function lengthOfVector([x, y]) {
  return Math.sqrt(x * x + y * y);
}

const dist = lengthOfVector(vector);  // dist = 5

function area({width, height}) {
  return width * height;
}
const a = area(dims);  // a = 45000
```

Use object declaration short cuts

old code

new code

javascript

```
const width = 300;
 const height = 150;
 const obj = {
   width: width,
   height: height,
   area: function() {
     return this.width * this.height
   },
 };
```

javascript

```
const width = 300;
 const height = 150;
 const obj = {
   width,
   height,
   area() {
     return this.width * this.height;
   },
 };
```

Use the rest parameter and the spread operator `...`

The rest parameter can be used to consume any number of parameters. Example

The spread operator can be used to expand an iterable into arguments

or copy an array

or to merge objects

javascript

```
function log(className, ...args) {
   const elem = document.createElement('div');
   elem.className = className;
   elem.textContent = args.join(' ');
   document.body.appendChild(elem);
 }
```

javascript

```
const position = [1, 2, 3];
someMesh.position.set(...position);
```

javascript

```
const copiedPositionArray = [...position];
copiedPositionArray.push(4); // [1,2,3,4]
console.log(position); // [1,2,3] position is unaffected
```

javascript

```
const a = {abc: 123};
const b = {def: 456};
const c = {...a, ...b};  // c is now {abc: 123, def: 456}
```

Use `class`

The syntax for making class like objects pre ES5 was unfamiliar to most programmers. As of ES5 you can now use the `class` keyword which is closer to the style of C++/C#/Java.

Understand getters and setters

Getters and setters are common in most modern languages. The `class` syntax of ES5 makes them much easier than pre ES5.

Use arrow functions where appropriate

This is especially useful with callbacks and promises.

Arrow functions bind `this` to the context in which you create the arrow function.

is a shortcut for

See link above for more info on `this`.

```javascript
loader.load((texture) => {
  // use texture
});
```

```javascript
const foo = (args) => {/* code */};
```

```javascript
const foo = (function(args) {/* code */}).bind(this));
```

Promises as well as async/await

Promises help with asynchronous code. Async/await help use promises.

It's too big a topic to go into here but you can read up on promises here and async/await here.

Use Template Literals

Template literals are strings using backticks instead of quotes.

Template literals have basically 2 features. One is they can be multi-line

`foo` and `bar` above are the same.

The other is that you can pop out of string mode and insert snippets of JavaScript using `${javascript-expression}`. This is the template part. Example:

or

or

```javascript
const foo = `this is a template literal`;
```

```javascript
const foo = `this
is
a
template
literal`;
const bar = "this\nis\na\ntemplate\nliteral";
```

```javascript
const r = 192;
const g = 255;
const b = 64;
const rgbCSSColor = `rgb(${r},${g},${b})`;
```

```javascript
const color = [192, 255, 64];
const rgbCSSColor = `rgb(${color.join(',')})`;
```

```javascript
const aWidth = 10;
const bWidth = 20;
someElement.style.width = `${aWidth + bWidth}px`;
```

Learn JavaScript coding conventions.

While you're welcome to format your code any way you chose there is at least one convention you should be aware of. Variables, function names, method names, in JavaScript are all lowerCasedCamelCase. Constructors, the names of classes are CapitalizedCamelCase. If you follow this rule your code will match most other JavaScript. Many linters, programs that check for obvious errors in your code, will point out errors if you use the wrong case since by following the convention above they can know when you're using something incorrectly.

```javascript
const v = new vector(); // clearly an error if all classes start with a capital let
const v = Vector();     // clearly an error if all functions start with a lowercase
```

Consider using Visual Studio Code

Of course use whatever editor you want but if you haven't tried it consider using Visual Studio Code for JavaScript and after installing it setup eslint. It might take a few minutes to setup but it will help you immensely with finding bugs in your JavaScript.

Some examples

If you enable the `no-undef` rule then VSCode via ESLint will warn you of many undefined variables.

Above you can see I mis-spelled `doTheThing` as `doThing`. There's a red squiggle under `doThing` and hovering over it it tells me it's undefined. One error avoided.

If you're using `<script>` tags to include three.js you'll get warnings using `THREE` so add `/* global THREE */` at the top of your JavaScript files to tell eslint that `THREE` exists. (or better, use `import` ☺)

Above you can see eslint knows the rule that `UpperCaseNames` are constructors and so you should be using `new`. Another error caught and avoided. This is the `new-cap` rule.

There are 100s of rules you can turn on or off or customize. For example above I mentioned you should use `const` and `let` over `var`.

Here I used `var` and it warned me I should use `let` or `const`

Here I used `let` but it saw I never change the value so it suggested I use `const`.

Of course if you'd prefer to keep using `var` you can just turn off that rule. As I said above though I prefer to use `const` and `let` over `var` as they just work better and prevent bugs.

For those cases where you really need to override a rule you can add comments to disable them for a single line or a section of code.

If you really need to support legacy browsers use a transpiler

Most modern browsers are auto-updated so using all these features will help you be productive and avoid bugs. That said, if you're on a project that absolutely must support old browsers there are tools that will take your ES5/ES6/ES7 code and transpile the code back to pre ES5 Javascript.

Installation

GUIDE

Source: https://threejs.org/manual/en/installation.html

This page covers how to set up a three.js project, including the basic project structure and two installation methods: installing via NPM with a build tool, or importing from a CDN.

Project structure

Every three.js project needs at least one HTML file to define the webpage, and a JavaScript file to run your three.js code. The structure and naming choices below aren't required, but will be used throughout this guide for consistency.

- *index.html*
- *main.js*
- *public/*

The *public/* folder is sometimes also called a "static" folder, because the files it contains are pushed to the website unchanged. Usually textures, audio, and 3D models will go here.

Now that we've set up the basic project structure, we need a way to run the project locally and access it through a web browser. Installation and local development can be

accomplished with npm and a build tool, or by importing three.js from a CDN. Both options are explained in the sections below.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My first three.js app</title>
    <style>
      body { margin: 0; }
    </style>
  </head>
  <body>
    <script type="module" src="/main.js"></script>
  </body>
</html>
```

main.js

```
import * as THREE from 'three';

...
```

Option 1: Install with NPM and a build tool

Installing from the npm package registry and using a build tool is the recommended approach for most users — the more dependencies your project needs, the more likely you are to run into problems that the static hosting cannot easily resolve. With a build tool, importing local JavaScript files and npm packages should work out of the box, without import maps.

Steps:

- Install Node.js. We'll need it to manage dependencies and to run our build tool.
- Install three.js and a build tool, Vite, using a terminal in your project folder. Vite will be used during development, but it isn't part of the final webpage. If you prefer to use another build tool, that's fine — we support modern build tools that can import ES Modules.

Installation added *node_modules/* and *package.json* to my project. What are they?

npm uses *package.json* to describe which versions of each dependency you've installed. If you have other people working on the project with you, they can install the

original versions of each dependency simply by running *npm install*. If you're using version history, commit *package.json*.

npm installs the code for each dependency in a new *node_modules/* folder. When Vite builds your application, it sees imports for 'three' and pulls three.js files automatically from this folder. The *node_modules/* folder is used only during development, and shouldn't be uploaded to your web hosting provider or committed to version history.

Using three.js with TypeScript: Community-maintained TypeScript type definitions for three.js are available at three-types/three-ts-types.

From your terminal, run:

What is *npx*? npx is installed with Node.js, and runs command line programs like Vite so that you don't have to search for the right file in *node_modules/* yourself. If you prefer, you can put Vite's common commands into the package.json:scripts list, and use *npm run dev* instead.

If everything went well, you'll see a URL like *http://localhost:5173* appear in your terminal, and can open that URL to see your web application.

The page will be blank — you're ready to create a scene.

If you want to learn more about these tools before you continue, see:

- three.js journey: Local Server
- Vite: Command Line Interface
- MDN: Package management basics

Later, when you're ready to deploy your web application, you'll just need to tell Vite to run a production build — *npx vite build*. Everything used by the application will be compiled, optimized, and copied into the *dist/* folder. The contents of that folder are ready to be hosted on your website.

shell

```
# three.js
npm install --save three

# vite
npm install --save-dev vite
```

```shell
npx vite
```

```shell
npx vite build
```

Option 2: Import from a CDN

Installing without build tools will require some changes to the project structure given above.

We imported code from 'three' (an npm package) in *main.js*, and web browsers don't know what that means. In *index.html* we'll need to add an import map defining where to get the package. Put the code below inside the tag, after the styles.

Don't forget to replace with an actual version of three.js, like *"v0.149.0"*. The most recent version can be found on the npm version list.

We'll also need to run a *local server* to host these files at URL where the web browser can access them. While it's technically possible to double-click an HTML file and open it in your browser, important features that we'll later implement, do not work when the page is opened this way, for security reasons.

Install Node.js, then run serve to start a local server in the project's directory:

If everything went well, you'll see a URL like http://localhost:3000 appear in your terminal, and can open that URL to see your web application.

The page will be blank — you're ready to create a scene.

Many other local static servers are available — some use different languages instead of Node.js, and others are desktop applications. They all work basically the same way, and we've provided a few alternatives below.

More local servers:

Command Line: Command line local servers run from a terminal window. The associated programming language may need to be installed first.

- *npx http-server* (Node.js)

- *npx five-server* (Node.js)
- *python -m SimpleHTTPServer* (Python 2.x)
- *python -m http.server* (Python 3.x)
- *php -S localhost:8000* (PHP 5.4+)

GUI: GUI local servers run as an application window on your computer, and may have a user interface.

- Servez

Code Editor Plugins: Some code editors have plugins that spawn a simple server on demand.

- Five Server for Visual Studio Code
- Live Server for Visual Studio Code
- Live Server for Atom

When you're ready to deploy your web application, push the source files to your web hosting provider — no need to build or compile anything. The downside of that tradeoff is that you'll need to be careful to keep the import map updated with any dependencies (and dependencies of dependencies!) that your application requires. If the CDN hosting your dependencies goes down temporarily, your website will stop working too.

IMPORTANT: Import all dependencies from the same version of three.js, and from the same CDN. Mixing files from different sources may cause duplicate code to be included, or even break the application in unexpected ways.

index.html

```
<script type="importmap">
{
  "imports": {
    "three": "https://cdn.jsdelivr.net/npm/three@<version>/build/three.module.js",
    "three/addons/": "https://cdn.jsdelivr.net/npm/three@<version>/examples/jsm/"
  }
}
</script>
```

shell

```
npx serve .
```

Addons

Out of the box, three.js includes the fundamentals of a 3D engine. Other three.js components — such as controls, loaders, and post-processing effects — are part of the addons/ directory. Addons do not need to be *installed* separately, but do need to be *imported* separately.

The example below shows how to import three.js with the `OrbitControls` and `GLTFLoader` addons. Where necessary, this will also be mentioned in each addon's documentation or examples.

Some excellent third-party projects are available for three.js, too. These need to be installed separately — see Libraries and Plugins.

main.js

```
import * as THREE from 'three';
import { OrbitControls } from 'three/addons/controls/OrbitControls.js';
import { GLTFLoader } from 'three/addons/loaders/GLTFLoader.js';

const controls = new OrbitControls( camera, renderer.domElement );
const loader = new GLTFLoader();
```

Next Steps

You're now ready to create a scene.

Setup

GUIDE

Source: https://threejs.org/manual/en/setup.html

A guide to setting up your computer as a development environment for three.js, including installing a local web server needed for WebGL development due to browser security restrictions.

Setup

This article is one in a series of articles about three.js. The first article was about three.js fundamentals. If you haven't read that yet you might want to start there.

Before we go any further we need to talk about setting up your computer as a development environment. In particular, for security reasons, WebGL cannot use images from your hard drive directly. That means in order to do development you

need to use a web server. Fortunately development web servers are super easy to setup and use.

First off if you'd like you can download this entire site from this link. Once downloaded double click the zip file to unpack the files.

Next download one of these simple web servers.

Servez (GUI option)

If you'd prefer a web server with a user interface there's Servez.

Just point it at the folder where you unzipped the files, click "Start", then go to in your browser http://localhost:8080/ or if you'd like to browse the samples go to http://localhost:8080/threejs.

To stop serving click stop or quit Servez.

Node.js servez (command line option)

If you prefer the command line (I do), another way is to use node.js. Download it, install it, then open a command prompt / console / terminal window. If you're on Windows the installer will add a special "Node Command Prompt" so use that.

Then install the servez by typing

```shell
npm -g install servez
```

OSX installation

If you're on OSX use

```shell
sudo npm -g install servez
```

Running servez

Once you've done that type

servez path/to/folder/where/you/unzipped/files

Or if you're like me

cd path/to/folder/where/you/unzipped/files servez

It should print something like

Then in your browser go to http://localhost:8080/.

If you don't specify a path then servez will serve the current folder.

If either of those options are not to your liking there are many other simple servers to choose from.

Now that you have a server setup we can move on to textures.

```shell
servez path/to/folder/where/you/unzipped/files
```

```shell
cd path/to/folder/where/you/unzipped/files
servez
```

Creating a scene

GUIDE

Source: https://threejs.org/manual/en/creating-a-scene.html

A brief introduction to three.js that walks through setting up a scene with a spinning cube, covering scene/camera/renderer setup, geometry, materials, meshes, and animation loops.

Before we start

The goal of this section is to give a brief introduction to three.js. We will start by setting up a scene, with a spinning cube. A working example is provided at the bottom of the page in case you get stuck and need help.

If you haven't yet, go through the `Installation` guide. We'll assume you've already set up the same project structure (including *index.html* and *main.js*), have installed

three.js, and are either running a build tool, or using a local server with a CDN and import maps.

Creating the scene

To actually be able to display anything with three.js, we need three things: scene, camera and renderer, so that we can render the scene with camera.

Let's take a moment to explain what's going on here. We have now set up the scene, our camera and the renderer.

There are a few different cameras in three.js. For now, let's use a `PerspectiveCamera`.

The first attribute is the `field of view`. FOV is the extent of the scene that is seen on the display at any given moment. The value is in degrees.

The second one is the `aspect ratio`. You almost always want to use the width of the element divided by the height, or you'll get the same result as when you play old movies on a widescreen TV - the image looks squished.

The next two attributes are the `near` and `far` clipping plane. What that means, is that objects further away from the camera than the value of `far` or closer than `near` won't be rendered. You don't have to worry about this now, but you may want to use other values in your apps to get better performance.

Next up is the renderer. In addition to creating the renderer instance, we also need to set the size at which we want it to render our app. It's a good idea to use the width and height of the area we want to fill with our app - in this case, the width and height of the browser window. For performance intensive apps, you can also give `setSize` smaller values, like `window.innerWidth/2` and `window.innerHeight/2`, which will make the app render at quarter size.

If you wish to keep the size of your app but render it at a lower resolution, you can do so by calling `setSize` with false as `updateStyle` (the third argument). For example, `setSize(window.innerWidth/2, window.innerHeight/2, false)` will render your app at

half resolution, given that your

Last but not least, we add the `renderer` element to our HTML document. This is a

"That's all good, but where's that cube you promised?" Let's add it now.

To create a cube, we need a `BoxGeometry`. This is an object that contains all the points (`vertices`) and fill (`faces`) of the cube. We'll explore this more in the future.

In addition to the geometry, we need a material to color it. Three.js comes with several materials, but we'll stick to the `MeshBasicMaterial` for now. All materials take an object of properties which will be applied to them. To keep things very simple, we only supply a color attribute of `0x00ff00`, which is green. This works the same way that colors work in CSS or Photoshop (`hex colors`).

The third thing we need is a `Mesh`. A mesh is an object that takes a geometry, and applies a material to it, which we then can insert to our scene, and move freely around.

By default, when we call `scene.add()`, the thing we add will be added to the coordinates `(0,0,0)`. This would cause both the camera and the cube to be inside each other. To avoid this, we simply move the camera out a bit.

main.js

```
import * as THREE from 'three';

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHei

const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );
```

javascript

```
const geometry = new THREE.BoxGeometry( 1, 1, 1 );
const material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
const cube = new THREE.Mesh( geometry, material );
scene.add( cube );

camera.position.z = 5;
```

Rendering the scene

If you copied the code from above into the main.js file we created earlier, you wouldn't be able to see anything. This is because we're not actually rendering anything yet. For that, we need what's called a render or animation loop.

This will create a loop that causes the renderer to draw the scene every time the screen is refreshed (on a typical screen this means 60 times per second). If you're new to writing games in the browser, you might say "why don't we just create a setInterval ?" The thing is - we could, but `requestAnimationFrame` which is internally used in `WebGLRenderer` has a number of advantages. Perhaps the most important one is that it pauses when the user navigates to another browser tab, hence not wasting their precious processing power and battery life.

javascript

```
function animate( time ) {
  renderer.render( scene, camera );
}
renderer.setAnimationLoop( animate );
```

Animating the cube

If you insert all the code above into the file you created before we began, you should see a green box. Let's make it all a little more interesting by rotating it.

Add the following code right above the `renderer.render` call in your `animate` function:

This will be run every frame (normally 60 times per second), and give the cube a nice rotation animation. Basically, anything you want to move or change while the app is running has to go through the animation loop. You can of course call other functions from there, so that you don't end up with an `animate` function that's hundreds of lines.

javascript

```
cube.rotation.x = time / 2000;
cube.rotation.y = time / 1000;
```

The result

Congratulations! You have now completed your first three.js application. It's simple, but you have to start somewhere.

The full code is available below and as an editable live example. Play around with it to get a better understanding of how it works.

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>My first three.js app</title>
    <style>
      body { margin: 0; }
    </style>
  </head>
  <body>
    <script type="module" src="/main.js"></script>
  </body>
</html>
```

main.js

```
import * as THREE from 'three';

const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHei

const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
renderer.setAnimationLoop( animate );
document.body.appendChild( renderer.domElement );

const geometry = new THREE.BoxGeometry( 1, 1, 1 );
const material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
const cube = new THREE.Mesh( geometry, material );
scene.add( cube );

camera.position.z = 5;

function animate( time ) {

  cube.rotation.x = time / 2000;
  cube.rotation.y = time / 1000;

  renderer.render( scene, camera );

}
```

Creating Text

GUIDE

Source: https://threejs.org/manual/en/creating-text.html

A guide covering multiple approaches to adding and rendering text in a three.js application, from simple DOM overlays to procedural 3D text geometries, bitmap fonts, and Troika Text.

1. DOM + CSS

Using HTML is generally the easiest and fastest manner to add text. This is the method used for descriptive overlays in most three.js examples.

You can add content to a

and use CSS markup to position absolutely at a position above all others with a z-index especially if you are running three.js full screen.

```
html
```

```
<div id="info">Description</div>
```

```css
#info {
  position: absolute;
  top: 10px;
  width: 100%;
  text-align: center;
  z-index: 100;
  display:block;
}
```

2. Use `CSS2DRenderer` or `CSS3DRenderer`

Use these renderers to draw high-quality text contained in DOM elements to your three.js scene. This is similar to 1. except that with these renderers elements can be integrated more tightly and dynamically into the scene.

3. Draw text to canvas and use as a `Texture`

Use this method if you wish to draw text easily on a plane in your three.js scene.

4. Create a model in your favourite 3D application and export to three.js

Use this method if you prefer working with your 3d applications and importing the models to three.js.

5. Procedural Text Geometry

If you prefer to work purely in THREE.js or to create procedural and dynamic 3D text geometries, you can create a mesh whose geometry is an instance of THREE.TextGeometry:

```
new THREE.TextGeometry( text, parameters );
```

In order for this to work, however, your TextGeometry will need an instance of THREE.Font to be set on its "font" parameter.

See the `TextGeometry` page for more info on how this can be done, descriptions of each accepted parameter, and a list of the JSON fonts that come with the THREE.js distribution itself.

Examples

WebGL / geometry / text

WebGL / shadowmap

If Typeface is down, or you want to use a font that is not there, there's a tutorial with a python script for blender that allows you to export text to Three.js's JSON format: http://www.jaanga.com/2012/03/blender-to-threejs-create-3d-text-with.html

6. Bitmap Fonts

BMFonts (bitmap fonts) allow batching glyphs into a single BufferGeometry. BMFont rendering supports word-wrapping, letter spacing, kerning, signed distance fields with standard derivatives, multi-channel signed distance fields, multi-texture fonts, and more. See three-mesh-ui or three-bmfont-text.

Stock fonts are available in projects like A-Frame Fonts, or you can create your own from any .TTF font, optimizing to include only characters required for a project.

Some helpful tools:

- msdf-bmfont-web *(web-based)*
- msdf-bmfont-xml *(commandline)*
- hiero *(desktop app)*

7. Troika Text

The troika-three-text package renders quality antialiased text using a similar technique as BMFonts, but works directly with any .TTF or .WOFF font file so you don't have to pregenerate a glyph texture offline. It also adds capabilities including:

- Effects like strokes, drop shadows, and curvature
- The ability to apply any three.js Material, even a custom ShaderMaterial
- Support for font ligatures, scripts with joined letters, and right-to-left/bidirectional layout
- Optimization for large amounts of dynamic text, performing most work off the main thread in a web worker

Drawing Lines

GUIDE

Source: https://threejs.org/manual/en/drawing-lines.html

A tutorial on how to draw lines in Three.js using LineBasicMaterial, BufferGeometry with vertices, and the Line object, rather than wireframe Mesh geometry.

Drawing Lines

Let's say you want to draw a line or a circle, not a wireframe `Mesh`. First we need to set up the renderer, scene and camera (see the Creating a scene page).

Here is the code that we will use:

Next thing we will do is define a material. For lines we have to use `LineBasicMaterial` or `LineDashedMaterial`.

After material we will need a geometry with some vertices:

Note that lines are drawn between each consecutive pair of vertices, but not between the first and last (the line is not closed.)

Now that we have points for two lines and a material, we can put them together to form a line.

All that's left is to add it to the scene and call `renderer.render()`.

You should now be seeing an arrow pointing upwards, made from two blue lines.

javascript

```
const renderer = new THREE.WebGLRenderer();
renderer.setSize( window.innerWidth, window.innerHeight );
document.body.appendChild( renderer.domElement );

const camera = new THREE.PerspectiveCamera( 45, window.innerWidth / window.innerHei
camera.position.set( 0, 0, 100 );
camera.lookAt( 0, 0, 0 );

const scene = new THREE.Scene();
```

javascript

```
//create a blue LineBasicMaterial
const material = new THREE.LineBasicMaterial( { color: 0x0000ff } );
```

javascript

```
const points = [];
points.push( new THREE.Vector3( - 10, 0, 0 ) );
points.push( new THREE.Vector3( 0, 10, 0 ) );
points.push( new THREE.Vector3( 10, 0, 0 ) );

const geometry = new THREE.BufferGeometry().setFromPoints( points );
```

javascript

```
const line = new THREE.Line( geometry, material );
```

javascript

```
scene.add( line );
renderer.render( scene, camera );
```

Fundamentals

Fundamentals

GUIDE

Source: https://threejs.org/manual/en/fundamentals.html

An introductory article covering the basics of three.js, including the structure of a three.js application, building a 'Hello Cube' scene with renderer, camera, geometry, material, and mesh, adding animation and lighting, and setting up ES6 modules with import maps.

Introduction to Three.js

This is the first article in a series of articles about three.js. Three.js is a 3D library that tries to make it as easy as possible to get 3D content on a webpage.

Three.js is often confused with WebGL since more often than not, but not always, three.js uses WebGL to draw 3D. WebGL is a very low-level system that only draws points, lines, and triangles. To do anything useful with WebGL generally requires quite a bit of code and that is where three.js comes in. It handles stuff like scenes, lights, shadows, materials, textures, 3d math, all things that you'd have to write yourself if you were to use WebGL directly.

These tutorials assume you already know JavaScript and, for the most part they will use ES6 style. Most browsers that support three.js are auto-updated so most users should be able to run this code. If you'd like to make this code run on really old browsers look into a transpiler like Babel. Of course users running really old browsers probably have machines that can't run three.js.

When learning most programming languages the first thing people do is make the computer print "Hello World!". For 3D one of the most common first things to do is to make a 3D cube. So let's start with "Hello Cube!"

Structure of a Three.js App

Before we get started let's try to give you an idea of the structure of a three.js app. A three.js app requires you to create a bunch of objects and connect them together.

Things to notice about the diagram:

- There is a Renderer. This is arguably the main object of three.js. You pass a Scene and a Camera to a Renderer and it renders (draws) the portion of the 3D scene that is inside the frustum of the camera as a 2D image to a canvas.

- There is a scenegraph which is a tree like structure, consisting of various objects like a Scene object, multiple Mesh objects, Light objects, Group, Object3D, and Camera objects. A Scene object defines the root of the scenegraph and contains properties like the background color and fog. These objects define a hierarchical parent/child tree like structure and represent where objects appear and how they are oriented. Children are positioned and oriented relative to their parent. For example the wheels on a car might be children of the car so that moving and orienting the car's object automatically moves the wheels.

Note in the diagram Camera is half in half out of the scenegraph. This is to represent that in three.js, unlike the other objects, a Camera does not have to be in the scenegraph to function. Just like other objects, a Camera, as a child of some other object, will move and orient relative to its parent object.

- Mesh objects represent drawing a specific Geometry with a specific Material. Both Material objects and Geometry objects can be used by multiple Mesh objects. For example to draw two blue cubes in different locations we could need two Mesh objects to represent the position and orientation of each cube. We would only need one Geometry to hold the vertex data for a cube and we would only need one Material to specify the color blue. Both Mesh objects could reference the same Geometry object and the same Material object.

- Geometry objects represent the vertex data of some piece of geometry like a sphere, cube, plane, dog, cat, human, tree, building, etc... Three.js provides many kinds of built in geometry primitives. You can also create custom geometry as well as load geometry from files.

- Material objects represent the surface properties used to draw geometry including things like the color to use and how shiny it is. A Material can also reference one or more Texture objects which can be used, for example, to wrap an image onto the surface of a geometry.

- Texture objects generally represent images either loaded from image files, generated from a canvas, or rendered from another scene.

- Light objects represent different kinds of lights.

Hello Cube Setup

Given all of that we're going to make the smallest "Hello Cube" setup.

First let's load three.js. It's important you put `type="module"` in the script tag. This enables us to use the `import` keyword to load three.js. As of r147, this is the only way to load three.js properly. Modules have the advantage that they can easily import other modules they need. That saves us from having to manually load extra scripts they are dependent on.

Next we need is a `<canvas>` tag. We will ask three.js to draw into that canvas so we need to look it up.

After we look up the canvas we create a WebGLRenderer. The renderer is the thing responsible for actually taking all the data you provide and rendering it to the canvas.

Note there are some esoteric details here. If you don't pass a canvas into three.js it will create one for you but then you have to add it to your document. Where to add it may change depending on your use case and you'll have to change your code so I find that passing a canvas to three.js feels a little more flexible. I can put the canvas anywhere and the code will find it whereas if I had code to insert the canvas into to the document I'd likely have to change that code if my use case changed.

```html
<script type="module">
import * as THREE from 'three';
</script>
```

```html
<body>
  <canvas id="c"></canvas>
</body>
```

```html
<script type="module">
import * as THREE from 'three';

function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
  ...
</script>
```

Camera Setup

Next up we need a camera. We'll create a PerspectiveCamera.

`fov` is short for `field of view`. In this case 75 degrees in the vertical dimension. Note that most angles in three.js are in radians but for some reason the perspective camera takes degrees.

`aspect` is the display aspect of the canvas. By default a canvas is 300x150 pixels which makes the aspect 300/150 or 2.

`near` and `far` represent the space in front of the camera that will be rendered. Anything before that range or after that range will be clipped (not drawn).

Those four settings define a "frustum". A frustum is the name of a 3d shape that is like a pyramid with the tip sliced off. In other words think of the word "frustum" as another 3D shape like sphere, cube, prism, frustum.

The height of the near and far planes are determined by the field of view. The width of both planes is determined by the field of view and the aspect.

Anything inside the defined frustum will be drawn. Anything outside will not.

The camera defaults to looking down the -Z axis with +Y up. We'll put our cube at the origin so we need to move the camera back a little from the origin in order to see anything.

javascript

```javascript
const fov = 75;
const aspect = 2;  // the canvas default
const near = 0.1;
const far = 5;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
```

javascript

```javascript
camera.position.z = 2;
```

Scene, Geometry, Material, and Mesh

Next we make a Scene. A Scene in three.js is the root of a form of scene graph. Anything you want three.js to draw needs to be added to the scene.

Next up we create a BoxGeometry which contains the data for a box. Almost anything we want to display in Three.js needs geometry which defines the vertices that make up our 3D object.

We then create a basic material and set its color. Colors can be specified using standard CSS style 6 digit hex color values.

We then create a Mesh. A Mesh in three.js represents the combination of three things:

- A Geometry (the shape of the object)

- A Material (how to draw the object, shiny or flat, what color, what texture(s) to apply. Etc.)
- The position, orientation, and scale of that object in the scene relative to its parent. In the code below that parent is the scene.

And finally we add that mesh to the scene. We can then render the scene by calling the renderer's render function and passing it the scene and the camera.

It's kind of hard to tell that is a 3D cube since we're viewing it directly down the -Z axis and the cube itself is axis aligned so we're only seeing a single face.

```javascript
const scene = new THREE.Scene();
```

```javascript
const boxWidth = 1;
const boxHeight = 1;
const boxDepth = 1;
const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
```

```javascript
const material = new THREE.MeshBasicMaterial({color: 0x44aa88});
```

```javascript
const cube = new THREE.Mesh(geometry, material);
```

```javascript
scene.add(cube);
```

```javascript
renderer.render(scene, camera);
```

Animating with requestAnimationFrame

Let's animate it spinning and hopefully that will make it clear it's being drawn in 3D. To animate it we'll render inside a render loop using requestAnimationFrame.

`requestAnimationFrame` is a request to the browser that you want to animate something. You pass it a function to be called. In our case that function is `render`. The browser will call your function and if you update anything related to the display of the page the browser will re-render the page. In our case we are calling three's `renderer.render` function which will draw our scene.

`requestAnimationFrame` passes the time since the page loaded to our function. That time is passed in milliseconds. I find it's much easier to work with seconds so here we're converting that to seconds.

We then set the cube's X and Y rotation to the current time. These rotations are in radians. There are 2 pi radians in a circle so our cube should turn around once on each axis in about 6.28 seconds.

We then render the scene and request another animation frame to continue our loop. Outside the loop we call `requestAnimationFrame` one time to start the loop.

It's a little better but it's still hard to see the 3d.

javascript

```
function render(time) {
  time *= 0.001;  // convert time to seconds

  cube.rotation.x = time;
  cube.rotation.y = time;

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
requestAnimationFrame(render);
```

Adding Lighting

What would help is to add some lighting so let's add a light. There are many kinds of lights in three.js which we'll go over in a future article. For now let's create a directional light.

Directional lights have a position and a target. Both default to 0, 0, 0. In our case we're setting the light's position to -1, 2, 4 so it's slightly on the left, above, and behind our camera. The target is still 0, 0, 0 so it will shine toward the origin.

We also need to change the material. The MeshBasicMaterial is not affected by lights. Let's change it to a MeshPhongMaterial which is affected by lights.

```javascript
const color = 0xFFFFFF;
const intensity = 3;
const light = new THREE.DirectionalLight(color, intensity);
light.position.set(-1, 2, 4);
scene.add(light);
```

```javascript
const material = new THREE.MeshPhongMaterial({color: 0x44aa88});  // greenish blue
```

Multiple Cubes

Just for the fun of it let's add 2 more cubes.

We'll use the same geometry for each cube but make a different material so each cube can be a different color.

First we'll make a function that creates a new material with the specified color. Then it creates a mesh using the specified geometry and adds it to the scene and sets its X position.

Then we'll call it 3 times with 3 different colors and X positions saving the Mesh instances in an array.

Finally we'll spin all 3 cubes in our render function. We compute a slightly different rotation for each one.

If you compare it to the top down diagram above you can see it matches our expectations. With cubes at X = -2 and X = +2 they are partially outside our frustum. They are also somewhat exaggeratedly warped since the field of view across the canvas is so extreme.

As you can see we have 3 Mesh objects each referencing the same BoxGeometry. Each Mesh references a unique MeshPhongMaterial so that each cube can have a different color.

I hope this short intro helps to get things started. Next up we'll cover making our code responsive so it is adaptable to multiple situations.

javascript

```
function makeInstance(geometry, color, x) {
  const material = new THREE.MeshPhongMaterial({color});

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  cube.position.x = x;

  return cube;
}
```

javascript

```
const cubes = [
  makeInstance(geometry, 0x44aa88,  0),
  makeInstance(geometry, 0x8844aa, -2),
  makeInstance(geometry, 0xaa8844,  2),
];
```

javascript

```
function render(time) {
  time *= 0.001;  // convert time to seconds

  cubes.forEach((cube, ndx) => {
    const speed = 1 + ndx * .1;
    const rot = time * speed;
    cube.rotation.x = rot;
    cube.rotation.y = rot;
  });

  ...
```

ES6 Modules, Three.js, and Folder Structure

As of version r147 the preferred way to use three.js is via es6 modules and import maps.

es6 modules can be loaded via the `import` keyword in a script or inline via a `<script type="module">` tag. Notice `'three'` specifier there. If you leave it as it is, it will likely produce an error. An import map should be used to tell the browser where to find three.js. Note that path specifier can start only with `./` or `../`.

To import addons like OrbitControls.js use the following. Don't forget to add addons to the import map. You can also use a CDN.

To conclude, the recommended way of using three.js is to set up an import map pointing to the three.js module file and addons, then use `<script type="module">` with import statements to load three.js and its addons.

```html
<script type="module">
import * as THREE from 'three';

...
</script>
```

```html
<script type="importmap">
{
  "imports": {
    "three": "./path/to/three.module.js"
  }
}
</script>
```

```javascript
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
```

```html
<script type="importmap">
{
  "imports": {
    "three": "./path/to/three.module.js",
    "three/addons/": "./different/path/to/examples/jsm/"
  }
}
</script>
```

```html
<script type="importmap">
{
  "imports": {
    "three": "https://cdn.jsdelivr.net/npm/three@<version>/build/three.module.js",
    "three/addons/": "https://cdn.jsdelivr.net/npm/three@<version>/examples/jsm/"
  }
}
</script>
```

```html
<script type="importmap">
{
  "imports": {
    "three": "./path/to/three.module.js",
    "three/addons/": "./different/path/to/examples/jsm/"
  }
}
</script>

<script type="module">
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';

...
</script>
```

Matrix Transformations

GUIDE

Source: https://threejs.org/manual/en/matrix-transformations.html

Explains how Three.js uses matrices to encode 3D transformations (translations, rotations, and scaling) for Object3D instances, including how to update an object's transformation manually or automatically.

Matrix Transformations

Three.js uses `matrices` to encode 3D transformations---translations (position), rotations, and scaling. Every instance of `Object3D` has a `matrix` which stores that object's position, rotation, and scale. This page describes how to update an object's transformation.

Convenience properties and `matrixAutoUpdate`

There are two ways to update an object's transformation:

- Modify the object's `position`, `quaternion`, and `scale` properties, and let three.js recompute the object's matrix from these properties:

By default, the `matrixAutoUpdate` property is set true, and the matrix will be automatically recalculated. If the object is static, or you wish to manually control when recalculation occurs, better performance can be obtained by setting the property false:

And after changing any properties, manually update the matrix:

- Modify the object's matrix directly. The `Matrix4` class has various methods for modifying the matrix:

Note that `matrixAutoUpdate` *must* be set to `false` in this case, and you should make sure *not* to call `updateMatrix`. Calling `updateMatrix` will clobber the manual changes made to the matrix, recalculating the matrix from `position`, `scale`, and so on.

```javascript
object.position.copy( start_position );
object.quaternion.copy( quaternion );
```

```javascript
object.matrixAutoUpdate = false;
```

```javascript
object.updateMatrix();
```

```javascript
object.matrix.makeRotationFromQuaternion( quaternion );
object.matrix.setPosition( start_position );
object.matrixAutoUpdate = false;
```

Object and world matrices

An object's matrix stores the object's transformation *relative* to the object's parent; to get the object's transformation in *world* coordinates, you must access the object's world matrix.

When either the parent or the child object's transformation changes, you can request that the child object's world matrix be updated by calling `object.updateMatrixWorld()`.

An object can be transformed via `applyMatrix4()`. Note: Under-the-hood, this method relies on `Matrix4.decompose()`, and not all matrices are decomposable in this way. For example, if an object has a non-uniformly scaled parent, then the object's world matrix may not be decomposable, and this method may not be appropriate.

Rotation and Quaternion

Three.js provides two ways of representing 3D rotations: Euler angles and Quaternions, as well as methods for converting between the two. Euler angles are subject to a problem called "gimbal lock," where certain configurations can lose a degree of freedom (preventing the object from being rotated about one axis). For this reason, object rotations are *always* stored in the object's quaternion.

Previous versions of the library included a `useQuaternion` property which, when set to false, would cause the object's matrix to be calculated from an Euler angle. This practice is deprecated---instead, you should use the `object.setRotationFromEuler()` method, which will update the quaternion.

Scene Graph

REFERENCE

Source: https://threejs.org/manual/en/scenegraph.html

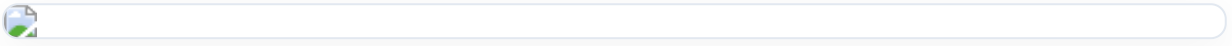
Extraction fallback content. [stop]

Scene Graph

Scene Graph

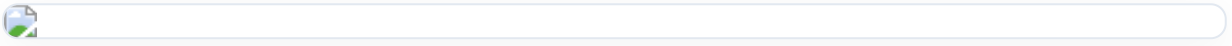
This article is part of a series of articles about three.js. The first article is three.js fundamentals. If you haven't read that yet you might want to consider starting there.

Three.js's core is arguably its scene graph. A scene graph in a 3D engine is a hierarchy of nodes in a graph where each node represents a local space.



That's kind of abstract so let's try to give some examples.

One example might be solar system, sun, earth, moon.



The Earth orbits the Sun. The Moon orbits the Earth. The Moon moves in a circle around the Earth. From the Moon's point of view it's rotating in the "local space" of the Earth. Even though its motion relative to the Sun is some crazy spirograph like curve

from the Moon's point of view it just has to concern itself with rotating around the Earth's local space.

To think of it another way, you living on the Earth do not have to think about the Earth's rotation on its axis nor its rotation around the Sun. You just walk or drive or swim or run as though the Earth is not moving or rotating at all. You walk, drive, swim, run, and live in the Earth's "local space" even though relative to the sun you are spinning around the earth at around 1000 miles per hour and around the sun at around 67,000 miles per hour. Your position in the solar system is similar to that of the moon above but you don't have to concern yourself. You just worry about your position relative to the earth in its "local space".

Let's take it one step at a time. Imagine we want to make a diagram of the sun, earth, and moon. We'll start with the sun by just making a sphere and putting it at the origin. Note: We're using sun, earth, moon as a demonstration of how to use a scene graph. Of course the real sun, earth, and moon use physics but for our purposes we'll fake it with a scene graph.

We're using a really low-polygon sphere. Only 6 subdivisions around its equator. This is so it's easy to see the rotation.

We're going to reuse the same sphere for everything so we'll set a scale for the sun mesh of 5x.

We also set the phong material's `emissive` property to yellow. A phong material's emissive property is basically the color that will be drawn with no light hitting the surface. Light is added to that color.

Let's also put a single point light in the center of the scene. We'll go into more details about point lights later but for now the simple version is a point light represents light that emanates from a single point.

To make it easy to see we're going to put the camera directly above the origin looking down. The easiest way to do that is to use the `lookAt` function. The `lookAt` function will orient the camera from its position to "look at" the position we pass to `lookAt`. Before we do that though we need to tell the camera which way the top of the camera is facing or rather which way is "up" for the camera. For most situations positive Y being up is good enough but since we are looking straight down we need to tell the camera that positive Z is up.

In the render loop, adapted from previous examples, we're rotating all objects in our `objects` array with this code.

Since we added the `sunMesh` to the `objects` array it will rotate.

click here to open in a separate window

Now let's add in the earth.

We make a material that is blue but we gave it a small amount of *emissive* blue so that it will show up against our black background.

We use the same `sphereGeometry` with our new blue `earthMaterial` to make an `earthMesh`. We position that 10 units to the left of the sun and add it to the scene. Since we added it to our `objects` array it will rotate too.

click here to open in a separate window

You can see both the sun and the earth are rotating but the earth is not going around the sun. Let's make the earth a child of the sun

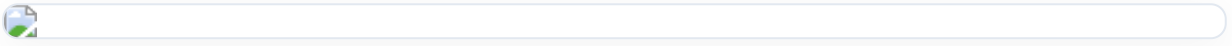
and...

click here to open in a separate window

What happened? Why is the earth the same size as the sun and why is it so far away? I actually had to move the camera from 50 units above to 150 units above to see the earth.

We made the `earthMesh` a child of the `sunMesh`. The `sunMesh` has its scale set to 5x with `sunMesh.scale.set(5, 5, 5)`. That means the `sunMesh`s local space is 5 times as big. Anything put in that space will be multiplied by 5. That means the earth is now 5x larger and its distance from the sun (`earthMesh.position.x = 10`) is also 5x as well.

Our scene graph currently looks like this



To fix it let's add an empty scene graph node. We'll parent both the sun and the earth to that node.

Here we made an `Object3D`. Like a `Mesh` it is also a node in the scene graph but unlike a `Mesh` it has no material or geometry. It just represents a local space.

Our new scene graph looks like this

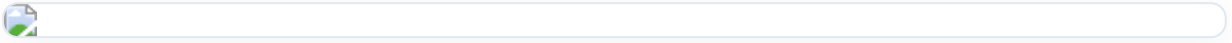
Both the `sunMesh` and the `earthMesh` are children of the `solarSystem`. All 3 are being rotated and now because the `earthMesh` is not a child of the `sunMesh` it is no longer scaled by 5x.

click here to open in a separate window

Much better. The earth is smaller than the sun and it's rotating around the sun and rotating itself.

Continuing that same pattern let's add a moon.

Again we added more invisible scene graph nodes. The first, an `Object3D` called `earthOrbit` and added both the `earthMesh` and the `moonOrbit` to it, also new. We then added the `moonMesh` to the `moonOrbit`. The new scene graph looks like this.



and here's that

click here to open in a separate window

You can see the moon follows the spirograph pattern shown at the top of this article but we didn't have to manually compute it. We just setup our scene graph to do it for us.

It is often useful to draw something to visualize the nodes in the scene graph. Three.js has some helpful ummmm, helpers to ummm, ... help with this.

One is called an `AxesHelper`. It draws 3 lines representing the local X, Y, and Z axes. Let's add one to every node we created.

On our case we want the axes to appear even though they are inside the spheres. To do this we set their material's `depthTest` to false which means they will not check to see if they are drawing behind something else. We also set their `renderOrder` to 1

(the default is 0) so that they get drawn after all the spheres. Otherwise a sphere might draw over them and cover them up.

click here to open in a separate window

We can see the x (red) and z (blue) axes. Since we are looking straight down and each of our objects is only rotating around its y axis we don't see much of the y (green) axes.

It might be hard to see some of them as there are 2 pairs of overlapping axes. Both the `sunMesh` and the `solarSystem` are at the same position. Similarly the `earthMesh` and `earthOrbit` are at the same position. Let's add some simple controls to allow us to turn them on/off for each node. While we're at it let's also add another helper called the `GridHelper`. It makes a 2D grid on the X,Z plane. By default the grid is 10x10 units.

We're also going to use lil-gui which is a UI library that is very popular with three.js projects. lil-gui takes an object and a property name on that object and based on the type of the property automatically makes a UI to manipulate that property.

We want to make both a `GridHelper` and an `AxesHelper` for each node. We need a label for each node so we'll get rid of the old loop and switch to calling some function to add the helpers for each node

`makeAxisGrid` makes an `AxisGridHelper` which is a class we'll create to make lil-gui happy. Like it says above lil-gui will automagically make a UI that manipulates the named property of some object. It will create a different UI depending on the type of property. We want it to create a checkbox so we need to specify a `bool` property. But, we want both the axes and the grid to appear/disappear based on a single property so we'll make a class that has a getter and setter for a property. That way we can let lil-gui think it's manipulating a single property but internally we can set the visible property of both the `AxesHelper` and `GridHelper` for a node.

One thing to notice is we set the `renderOrder` of the `AxesHelper` to 2 and for the `GridHelper` to 1 so that the axes get drawn after the grid. Otherwise the grid might overwrite the axes.

click here to open in a separate window

Turn on the `solarSystem` and you'll see how the earth is exactly 10 units out from the center just like we set above. You can see how the earth is in the *local space* of the

`solarSystem`. Similarly if you turn on the `earthOrbit` you'll see how the moon is exactly 2 units from the center of the *local space* of the `earthOrbit`.

A few more examples of scene graphs. An automobile in a simple game world might have a scene graph like this

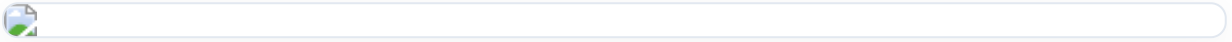
If you move the car's body all the wheels will move with it. If you wanted the body to bounce separate from the wheels you might parent the body and the wheels to a "frame" node that represents the car's frame.

Another example is a human in a game world.

You can see the scene graph gets pretty complex for a human. In fact that scene graph above is simplified. For example you might extend it to cover every finger (at least another 28 nodes) and every toe (yet another 28 nodes) plus ones for the face and jaw, the eyes and maybe more.

Let's make one semi-complex scene graph. We'll make a tank. The tank will have 6 wheels and a turret. The tank will follow a path. There will be a sphere that moves around and the tank will target the sphere.

Here's the scene graph. The meshes are colored in green, the `Object3D`s in blue, the lights in gold, and the cameras in purple. One camera has not been added to the scene graph.



Look in the code to see the setup of all of these nodes.

For the target, the thing the tank is aiming at, there is a `targetOrbit` (`Object3D`) which just rotates similar to the `earthOrbit` above. A `targetElevation` (`Object3D`) which is a child of the `targetOrbit` provides an offset from the `targetOrbit` and a base elevation. Childed to that is another `Object3D` called `targetBob` which just bobs up and down relative to the `targetElevation`. Finally there's the `targetMesh` which is just a cube we rotate and change its colors

For the tank there's an `Object3D` called `tank` which is used to move everything below it around. The code uses a `SplineCurve` which it can ask for positions along that curve. 0.0 is the start of the curve. 1.0 is the end of the curve. It asks for the current position where it puts the tank. It then asks for a position slightly further down the curve and uses that to point the tank in that direction using `Object3D.lookAt`.

The turret on top of the tank is moved automatically by being a child of the tank. To point it at the target we just ask for the target's world position and then again use `Object3D.lookAt`

There's a `turretCamera` which is a child of the `turretMesh` so it will move up and down and rotate with the turret. We make that aim at the target.

There is also a `targetCameraPivot` which is a child of `targetBob` so it floats around with the target. We aim that back at the tank. Its purpose is to allow the `targetCamera` to be offset from the target itself. If we instead made the camera a child of `targetBob` and just aimed the camera itself it would be inside the target.

Finally we rotate all the wheels

For the cameras we setup an array of all 4 cameras at init time with descriptions.

and cycle through our cameras at render time.

click here to open in a separate window

I hope this gives some idea of how scene graphs work and how you might use them. Making `Object3D` nodes and parenting things to them is an important step to using a 3D engine like three.js well. Often it might seem like some complex math is necessary to make something move and rotate the way you want. For example without a scene graph computing the motion of the moon or where to put the wheels of the car relative to its body would be very complicated but using a scene graph it becomes much easier.

Next up we'll go over materials.

```
js
```

```
// an array of objects whose rotation to update
const objects = [];

// use just one sphere for everything
const radius = 1;
const widthSegments = 6;
const heightSegments = 6;
const sphereGeometry = new THREE.SphereGeometry(
    radius, widthSegments, heightSegments);

const sunMaterial = new THREE.MeshPhongMaterial({emissive: 0xFFFF00});
const sunMesh = new THREE.Mesh(sphereGeometry, sunMaterial);
sunMesh.scale.set(5, 5, 5);  // make the sun large
scene.add(sunMesh);
objects.push(sunMesh);
```

js

```
{
  const color = 0xFFFFFF;
  const intensity = 500;
  const light = new THREE.PointLight(color, intensity);
  scene.add(light);
}
```

js

```
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
camera.position.set(0, 50, 0);
camera.up.set(0, 0, 1);
camera.lookAt(0, 0, 0);
```

js

```
objects.forEach((obj) => {
  obj.rotation.y = time;
});
```

js

```
const earthMaterial = new THREE.MeshPhongMaterial({color: 0x2233FF, emissive: 0x112
const earthMesh = new THREE.Mesh(sphereGeometry, earthMaterial);
earthMesh.position.x = 10;
scene.add(earthMesh);
objects.push(earthMesh);
```

js

```
-scene.add(earthMesh);
+sunMesh.add(earthMesh);
```

js

```
+const solarSystem = new THREE.Object3D();
+scene.add(solarSystem);
+objects.push(solarSystem);

const sunMaterial = new THREE.MeshPhongMaterial({emissive: 0xFFFF00});
const sunMesh = new THREE.Mesh(sphereGeometry, sunMaterial);
sunMesh.scale.set(5, 5, 5);
-scene.add(sunMesh);
+solarSystem.add(sunMesh);
objects.push(sunMesh);

const earthMaterial = new THREE.MeshPhongMaterial({color: 0x2233FF, emissive: 0x112
const earthMesh = new THREE.Mesh(sphereGeometry, earthMaterial);
earthMesh.position.x = 10;
-sunMesh.add(earthMesh);
+solarSystem.add(earthMesh);
objects.push(earthMesh);
```

js

```
+const earthOrbit = new THREE.Object3D();
+earthOrbit.position.x = 10;
+solarSystem.add(earthOrbit);
+objects.push(earthOrbit);

const earthMaterial = new THREE.MeshPhongMaterial({color: 0x2233FF, emissive: 0x112
const earthMesh = new THREE.Mesh(sphereGeometry, earthMaterial);
-earthMesh.position.x = 10; // note that this offset is already set in its parent's
-solarSystem.add(earthMesh);
+earthOrbit.add(earthMesh);
objects.push(earthMesh);

+const moonOrbit = new THREE.Object3D();
+moonOrbit.position.x = 2;
+earthOrbit.add(moonOrbit);

+const moonMaterial = new THREE.MeshPhongMaterial({color: 0x888888, emissive: 0x222
+const moonMesh = new THREE.Mesh(sphereGeometry, moonMaterial);
+moonMesh.scale.set(.5, .5, .5);
+moonOrbit.add(moonMesh);
+objects.push(moonMesh);
```

js

```
// add an AxesHelper to each node
objects.forEach((node) => {
  const axes = new THREE.AxesHelper();
  axes.material.depthTest = false;
  axes.renderOrder = 1;
  node.add(axes);
});
```

js

```
-// add an AxesHelper to each node
-objects.forEach((node) => {
-  const axes = new THREE.AxesHelper();
-  axes.material.depthTest = false;
-  axes.renderOrder = 1;
-  node.add(axes);
-});

+function makeAxisGrid(node, label, units) {
+  const helper = new AxisGridHelper(node, units);
+  gui.add(helper, 'visible').name(label);
+}
+
+makeAxisGrid(solarSystem, 'solarSystem', 25);
+makeAxisGrid(sunMesh, 'sunMesh');
+makeAxisGrid(earthOrbit, 'earthOrbit');
+makeAxisGrid(earthMesh, 'earthMesh');
+makeAxisGrid(moonOrbit, 'moonOrbit');
+makeAxisGrid(moonMesh, 'moonMesh');
```

js

```
visible
```

js

```
// move target
targetOrbit.rotation.y = time * .27;
targetBob.position.y = Math.sin(time * 2) * 4;
targetMesh.rotation.x = time * 7;
targetMesh.rotation.y = time * 13;
targetMaterial.emissive.setHSL(time * 10 % 1, 1, .25);
targetMaterial.color.setHSL(time * 10 % 1, 1, .25);
```

js

```
const tankPosition = new THREE.Vector2();
const tankTarget = new THREE.Vector2();

...

// move tank
const tankTime = time * .05;
curve.getPointAt(tankTime % 1, tankPosition);
curve.getPointAt((tankTime + 0.01) % 1, tankTarget);
tank.position.set(tankPosition.x, 0, tankPosition.y);
tank.lookAt(tankTarget.x, 0, tankTarget.y);
```

js

```
const targetPosition = new THREE.Vector3();

...

// face turret at target
targetMesh.getWorldPosition(targetPosition);
turretPivot.lookAt(targetPosition);
```

js

```
// make the turretCamera look at target
turretCamera.lookAt(targetPosition);
```

js

```
// make the targetCameraPivot look at the tank
tank.getWorldPosition(targetPosition);
targetCameraPivot.lookAt(targetPosition);
```

js

```
wheelMeshes.forEach((obj) => {
  obj.rotation.x = time * 3;
});
```

js

```
const cameras = [
  { cam: camera, desc: 'detached camera', },
  { cam: turretCamera, desc: 'on turret looking at target', },
  { cam: targetCamera, desc: 'near target looking at tank', },
  { cam: tankCamera, desc: 'above back of tank', },
];

const infoElem = document.querySelector('#info');
```

```
js
```

```
const camera = cameras[time * .25 % cameras.length | 0];
infoElem.textContent = camera.desc;
```

Primitives

GUIDE

Source: https://threejs.org/manual/en/primitives.html

This article covers the built-in 3D primitive geometries available in three.js, including shapes like boxes, spheres, cones, and more advanced options like extrusion, text, and wireframes. It also walks through a complete example demonstrating how to instantiate these primitives with random colors in a scene.

Primitives

Three.js has a large number of primitives. Primitives are generally 3D shapes that are generated at runtime with a bunch of parameters.

It's common to use primitives for things like a sphere for a globe or a bunch of boxes to draw a 3D graph. It's especially common to use primitives to experiment and get started with 3D. For the majority of 3D apps it's more common to have an artist make 3D models in a 3D modeling program like Blender or Maya or Cinema 4D. Later in this series we'll cover making and loading data from several 3D modeling programs. For now let's go over some of the available primitives.

Many of the primitives below have defaults for some or all of their parameters so you can use more or less depending on your needs.

BoxGeometry

A Box

javascript

```
const width = 8;  // ui: width
const height = 8;  // ui: height
const depth = 8;  // ui: depth
const geometry = new THREE.BoxGeometry( width, height, depth );
```

javascript

```
const width = 8;  // ui: width
const height = 8;  // ui: height
const depth = 8;  // ui: depth
const widthSegments = 4;  // ui: widthSegments
const heightSegments = 4;  // ui: heightSegments
const depthSegments = 4;  // ui: depthSegments
const geometry = new THREE.BoxGeometry(
        width, height, depth,
        widthSegments, heightSegments, depthSegments );
```

CircleGeometry

A flat circle

javascript

```
const radius = 7;  // ui: radius
const segments = 24;  // ui: segments
const geometry = new THREE.CircleGeometry( radius, segments );
```

javascript

```
const radius = 7;  // ui: radius
const segments = 24;  // ui: segments
const thetaStart = Math.PI * 0.25;  // ui: thetaStart
const thetaLength = Math.PI * 1.5;  // ui: thetaLength
const geometry = new THREE.CircleGeometry(
        radius, segments, thetaStart, thetaLength );
```

ConeGeometry

A Cone

javascript

```
const radius = 6;  // ui: radius
const height = 8;  // ui: height
const radialSegments = 16;  // ui: radialSegments
const geometry = new THREE.ConeGeometry( radius, height, radialSegments );
```

javascript

```
const radius = 6;  // ui: radius
const height = 8;  // ui: height
const radialSegments = 16;  // ui: radialSegments
const heightSegments = 2;  // ui: heightSegments
const openEnded = true;  // ui: openEnded
const thetaStart = Math.PI * 0.25;  // ui: thetaStart
const thetaLength = Math.PI * 1.5;  // ui: thetaLength
const geometry = new THREE.ConeGeometry(
        radius, height,
        radialSegments, heightSegments,
        openEnded,
        thetaStart, thetaLength );
```

CylinderGeometry

A Cylinder

javascript

```
const radiusTop = 4;  // ui: radiusTop
const radiusBottom = 4;  // ui: radiusBottom
const height = 8;  // ui: height
const radialSegments = 12;  // ui: radialSegments
const geometry = new THREE.CylinderGeometry(
        radiusTop, radiusBottom, height, radialSegments );
```

javascript

```
const radiusTop = 4;  // ui: radiusTop
const radiusBottom = 4;  // ui: radiusBottom
const height = 8;  // ui: height
const radialSegments = 12;  // ui: radialSegments
const heightSegments = 2;  // ui: heightSegments
const openEnded = false;  // ui: openEnded
const thetaStart = Math.PI * 0.25;  // ui: thetaStart
const thetaLength = Math.PI * 1.5;  // ui: thetaLength
const geometry = new THREE.CylinderGeometry(
        radiusTop, radiusBottom, height,
        radialSegments, heightSegments,
        openEnded,
        thetaStart, thetaLength );
```

DodecahedronGeometry

A dodecahedron (12 sides)

javascript

```
const radius = 7;  // ui: radius
const geometry = new THREE.DodecahedronGeometry( radius );
```

javascript

```
const radius = 7;  // ui: radius
const detail = 2;  // ui: detail
const geometry = new THREE.DodecahedronGeometry( radius, detail );
```

ExtrudeGeometry

An extruded 2d shape with optional bevelling. Here we are extruding a heart shape. Note this is the basis for TextGeometry.

javascript

```
const shape = new THREE.Shape();
const x = -2.5;
const y = -5;
shape.moveTo(x + 2.5, y + 2.5);
shape.bezierCurveTo(x + 2.5, y + 2.5, x + 2, y, x, y);
shape.bezierCurveTo(x - 3, y, x - 3, y + 3.5, x - 3, y + 3.5);
shape.bezierCurveTo(x - 3, y + 5.5, x - 1.5, y + 7.7, x + 2.5, y + 9.5);
shape.bezierCurveTo(x + 6, y + 7.7, x + 8, y + 4.5, x + 8, y + 3.5);
shape.bezierCurveTo(x + 8, y + 3.5, x + 8, y, x + 5, y);
shape.bezierCurveTo(x + 3.5, y, x + 2.5, y + 2.5, x + 2.5, y + 2.5);

const extrudeSettings = {
  steps: 2,  // ui: steps
  depth: 2,  // ui: depth
  bevelEnabled: true,  // ui: bevelEnabled
  bevelThickness: 1,  // ui: bevelThickness
  bevelSize: 1,  // ui: bevelSize
  bevelSegments: 2,  // ui: bevelSegments
};

const geometry = THREE.ExtrudeGeometry(shape, extrudeSettings);
```

javascript

```
const outline = new THREE.Shape([
  [ -2, -0.1], [  2, -0.1], [ 2,  0.6],
  [1.6,  0.6], [1.6,  0.1], [-2,  0.1],
].map(p => new THREE.Vector2(...p)));

const x = -2.5;
const y = -5;
const shape = new THREE.CurvePath();
const points = [
  [x + 2.5, y + 2.5],
  [x + 2.5, y + 2.5], [x + 2,   y      ], [x,       y      ],
  [x - 3,   y      ], [x - 3,   y + 3.5], [x - 3,   y + 3.5],
  [x - 3,   y + 5.5], [x - 1.5, y + 7.7], [x + 2.5, y + 9.5],
  [x + 6,   y + 7.7], [x + 8,   y + 4.5], [x + 8,   y + 3.5],
  [x + 8,   y + 3.5], [x + 8,   y      ], [x + 5,   y      ],
  [x + 3.5, y      ], [x + 2.5, y + 2.5], [x + 2.5, y + 2.5],
].map(p => new THREE.Vector3(...p, 0));
for (let i = 0; i < points.length; i += 3) {
  shape.add(new THREE.CubicBezierCurve3(...points.slice(i, i + 4)));
}

const extrudeSettings = {
  steps: 100,  // ui: steps
  bevelEnabled: false,
  extrudePath: shape,
};

const geometry =  new THREE.ExtrudeGeometry(outline, extrudeSettings);
return geometry;
```

IcosahedronGeometry

An icosahedron (20 sides)

javascript

```
const radius = 7;  // ui: radius
const geometry = new THREE.IcosahedronGeometry( radius );
```

javascript

```
const radius = 7;  // ui: radius
const detail = 2;  // ui: detail
const geometry = new THREE.IcosahedronGeometry( radius, detail );
```

LatheGeometry

A shape generated by spinning a line. Examples would be: lamps, bowling pins, candles, candle holders, wine glasses, drinking glasses, etc... You provide the 2d silhouette as series of points and then tell three.js how many subdivisions to make as it spins the silhouette around an axis.

javascript

```
const points = [];
for ( let i = 0; i < 10; ++ i ) {

        points.push( new THREE.Vector2( Math.sin( i * 0.2 ) * 3 + 3, ( i - 5 ) * .8

}

const geometry = new THREE.LatheGeometry( points );
```

javascript

```
const points = [];
for ( let i = 0; i < 10; ++ i ) {

        points.push( new THREE.Vector2( Math.sin( i * 0.2 ) * 3 + 3, ( i - 5 ) * .8

}

const segments = 12;  // ui: segments
const phiStart = Math.PI * 0.25;  // ui: phiStart
const phiLength = Math.PI * 1.5;  // ui: phiLength
const geometry = new THREE.LatheGeometry(
        points, segments, phiStart, phiLength );
```

OctahedronGeometry

An Octahedron (8 sides)

javascript

```
const radius = 7;  // ui: radius
const geometry = new THREE.OctahedronGeometry( radius );
```

javascript

```
const radius = 7;  // ui: radius
const detail = 2;  // ui: detail
const geometry = new THREE.OctahedronGeometry( radius, detail );
```

ParametricGeometry

A surface generated by providing a function that takes a 2D point from a grid and returns the corresponding 3d point.

javascript

```
const slices = 25;  // ui: slices
const stacks = 25;  // ui: stacks

// from: https://github.com/mrdoob/three.js/blob/b8d8a8625465bd634aa68e5846354d69f3
function klein( v, u, target ) {

        u *= Math.PI;
        v *= 2 * Math.PI;
        u = u * 2;

        let x;
        let z;

        if ( u < Math.PI ) {

                x = 3 * Math.cos( u ) * ( 1 + Math.sin( u ) ) + ( 2 * ( 1 - Math.cc
                z = - 8 * Math.sin( u ) - 2 * ( 1 - Math.cos( u ) / 2 ) * Math.sin(

        } else {

                x = 3 * Math.cos( u ) * ( 1 + Math.sin( u ) ) + ( 2 * ( 1 - Math.cc
                z = - 8 * Math.sin( u );

        }

        const y = - 2 * ( 1 - Math.cos( u ) / 2 ) * Math.sin( v );

        target.set( x, y, z ).multiplyScalar( 0.75 );

}

return new ParametricGeometry(
        klein, slices, stacks );
```

PlaneGeometry

A 2D plane

javascript

```
const width = 9;  // ui: width
const height = 9;  // ui: height
const geometry = new THREE.PlaneGeometry( width, height );
```

javascript

```
const width = 9;  // ui: width
const height = 9;  // ui: height
const widthSegments = 2;  // ui: widthSegments
const heightSegments = 2;  // ui: heightSegments
const geometry = new THREE.PlaneGeometry(
        width, height,
        widthSegments, heightSegments );
```

PolyhedronGeometry

Takes a set of triangles centered around a point and projects them onto a sphere

```javascript
const verticesOfCube = [
        - 1, - 1, - 1, 1, - 1, - 1, 1, 1, - 1, - 1, 1, - 1,
        - 1, - 1, 1, 1, - 1, 1, 1, 1, 1, - 1, 1, 1,
];
const indicesOfFaces = [
        2, 1, 0, 0, 3, 2,
        0, 4, 7, 7, 3, 0,
        0, 1, 5, 5, 4, 0,
        1, 2, 6, 6, 5, 1,
        2, 3, 7, 7, 6, 2,
        4, 5, 6, 6, 7, 4,
];
const radius = 7;  // ui: radius
const detail = 2;  // ui: detail
const geometry = new THREE.PolyhedronGeometry(
        verticesOfCube, indicesOfFaces, radius, detail );
```

RingGeometry

A 2D disc with a hole in the center

```javascript
const innerRadius = 2;  // ui: innerRadius
const outerRadius = 7;  // ui: outerRadius
const thetaSegments = 18;  // ui: thetaSegments
const geometry = new THREE.RingGeometry(
        innerRadius, outerRadius, thetaSegments );
```

```javascript
const innerRadius = 2;  // ui: innerRadius
const outerRadius = 7;  // ui: outerRadius
const thetaSegments = 18;  // ui: thetaSegments
const phiSegments = 2;  // ui: phiSegments
const thetaStart = Math.PI * 0.25;  // ui: thetaStart
const thetaLength = Math.PI * 1.5;  // ui: thetaLength
const geometry = new THREE.RingGeometry(
        innerRadius, outerRadius,
        thetaSegments, phiSegments,
        thetaStart, thetaLength );
```

ShapeGeometry

A 2D outline that gets triangulated

javascript

```
const shape = new THREE.Shape();
const x = - 2.5;
const y = - 5;
shape.moveTo( x + 2.5, y + 2.5 );
shape.bezierCurveTo( x + 2.5, y + 2.5, x + 2, y, x, y );
shape.bezierCurveTo( x - 3, y, x - 3, y + 3.5, x - 3, y + 3.5 );
shape.bezierCurveTo( x - 3, y + 5.5, x - 1.5, y + 7.7, x + 2.5, y + 9.5 );
shape.bezierCurveTo( x + 6, y + 7.7, x + 8, y + 4.5, x + 8, y + 3.5 );
shape.bezierCurveTo( x + 8, y + 3.5, x + 8, y, x + 5, y );
shape.bezierCurveTo( x + 3.5, y, x + 2.5, y + 2.5, x + 2.5, y + 2.5 );
const geometry = new THREE.ShapeGeometry( shape );
```

javascript

```
const shape = new THREE.Shape();
const x = - 2.5;
const y = - 5;
shape.moveTo( x + 2.5, y + 2.5 );
shape.bezierCurveTo( x + 2.5, y + 2.5, x + 2, y, x, y );
shape.bezierCurveTo( x - 3, y, x - 3, y + 3.5, x - 3, y + 3.5 );
shape.bezierCurveTo( x - 3, y + 5.5, x - 1.5, y + 7.7, x + 2.5, y + 9.5 );
shape.bezierCurveTo( x + 6, y + 7.7, x + 8, y + 4.5, x + 8, y + 3.5 );
shape.bezierCurveTo( x + 8, y + 3.5, x + 8, y, x + 5, y );
shape.bezierCurveTo( x + 3.5, y, x + 2.5, y + 2.5, x + 2.5, y + 2.5 );
const curveSegments = 5;  // ui: curveSegments
const geometry = new THREE.ShapeGeometry( shape, curveSegments );
```

SphereGeometry

A sphere

javascript

```
const radius = 7;  // ui: radius
const widthSegments = 12;  // ui: widthSegments
const heightSegments = 8;  // ui: heightSegments
const geometry = new THREE.SphereGeometry( radius, widthSegments, heightSegments );
```

javascript

```
const radius = 7;  // ui: radius
const widthSegments = 12;  // ui: widthSegments
const heightSegments = 8;  // ui: heightSegments
const phiStart = Math.PI * 0.25;  // ui: phiStart
const phiLength = Math.PI * 1.5;  // ui: phiLength
const thetaStart = Math.PI * 0.25;  // ui: thetaStart
const thetaLength = Math.PI * 0.5;  // ui: thetaLength
const geometry = new THREE.SphereGeometry(
        radius,
        widthSegments, heightSegments,
        phiStart, phiLength,
        thetaStart, thetaLength );
```

TetrahedronGeometry

A tetrahedron (4 sides)

javascript

```
const radius = 7;  // ui: radius
const geometry = new THREE.TetrahedronGeometry( radius );
```

javascript

```
const radius = 7;  // ui: radius
const detail = 2;  // ui: detail
const geometry = new THREE.TetrahedronGeometry( radius, detail );
```

TextGeometry

3D text generated from a 3D font and a string

javascript

```
const loader = new THREE.FontLoader();

loader.load('../resources/threejs/fonts/helvetiker_regular.typeface.json', (font) =
  const text = 'three.js';  // ui: text
  const geometry = new THREE.TextGeometry(text, {
    font: font,
    size: 3,  // ui: size
    depth: 0.2,  // ui: depth
    curveSegments: 12,  // ui: curveSegments
    bevelEnabled: true,  // ui: bevelEnabled
    bevelThickness: 0.15,  // ui: bevelThickness
    bevelSize: 0.3,  // ui: bevelSize
    bevelSegments: 5,  // ui: bevelSegments
  });
  ...
});
```

TorusGeometry

A torus (donut)

```javascript
const radius = 5;  // ui: radius
const tubeRadius = 2;  // ui: tubeRadius
const radialSegments = 8;  // ui: radialSegments
const tubularSegments = 24;  // ui: tubularSegments
const geometry = new THREE.TorusGeometry(
        radius, tubeRadius,
        radialSegments, tubularSegments );
```

TorusKnotGeometry

A torus knot

```javascript
const radius = 3.5;  // ui: radius
const tubeRadius = 1.5;  // ui: tubeRadius
const radialSegments = 8;  // ui: radialSegments
const tubularSegments = 64;  // ui: tubularSegments
const p = 2;  // ui: p
const q = 3;  // ui: q
const geometry = new THREE.TorusKnotGeometry(
        radius, tubeRadius, tubularSegments, radialSegments, p, q );
```

TubeGeometry

A circle traced down a path

```javascript
```

```
class CustomSinCurve extends THREE.Curve {

        constructor( scale ) {

                super();
                this.scale = scale;

        }
        getPoint( t ) {

                const tx = t * 3 - 1.5;
                const ty = Math.sin( 2 * Math.PI * t );
                const tz = 0;
                return new THREE.Vector3( tx, ty, tz ).multiplyScalar( this.scale )

        }

}

const path = new CustomSinCurve( 4 );
const tubularSegments = 20;  // ui: tubularSegments
const radius = 1;  // ui: radius
const radialSegments = 8;  // ui: radialSegments
const closed = false;  // ui: closed
const geometry = new THREE.TubeGeometry(
        path, tubularSegments, radius, radialSegments, closed );
```

EdgesGeometry

A helper object that takes another geometry as input and generates edges only if the angle between faces is greater than some threshold. For example if you look at the box at the top it shows a line going through each face showing every triangle that makes the box. Using an EdgesGeometry instead the middle lines are removed. Adjust the thresholdAngle below and you'll see the edges below that threshold disappear.

javascript

```
const size = 8;
const widthSegments = 2;
const heightSegments = 2;
const depthSegments = 2;
const boxGeometry = new THREE.BoxGeometry(
    size, size, size,
    widthSegments, heightSegments, depthSegments);
const geometry = new THREE.EdgesGeometry(boxGeometry);
```

javascript

```
const radius = 7;
const widthSegments = 6;
const heightSegments = 3;
const sphereGeometry = new THREE.SphereGeometry(
    radius, widthSegments, heightSegments);
const thresholdAngle = 1;  // ui: thresholdAngle
const geometry = new THREE.EdgesGeometry(sphereGeometry, thresholdAngle);
```

WireframeGeometry

Generates geometry that contains one line segment (2 points) per edge in the given geometry. Without this you'd often be missing edges or get extra edges since WebGL generally requires 2 points per line segment. For example if all you had was a single triangle there would only be 3 points. If you tried to draw it using a material with wireframe: true you would only get a single line. Passing that triangle geometry to a WireframeGeometry will generate a new geometry that has 3 lines segments using 6 points.

```
javascript
```

```
const size = 8;
const widthSegments = 2;  // ui: widthSegments
const heightSegments = 2;  // ui: heightSegments
const depthSegments = 2;  // ui: depthSegments
const geometry = new THREE.WireframeGeometry(
    new THREE.BoxGeometry(
      size, size, size,
      widthSegments, heightSegments, depthSegments));
```

Building an Example

We'll go over creating custom geometry in another article. For now let's make an example creating each type of primitive. We'll start with the examples from the previous article.

Near the top let's set a background color

This tells three.js to clear to lightish gray.

The camera needs to change position so that we can see all the objects.

Let's add a function, addObject, that takes an x, y position and an Object3D and adds the object to the scene.

Let's also make a function to create a random colored material. We'll use a feature of Color that lets you set a color based on hue, saturation, and luminance.

hue goes from 0 to 1 around the color wheel with red at 0, green at .33 and blue at .66. saturation goes from 0 to 1 with 0 having no color and 1 being most saturated. luminance goes from 0 to 1 with 0 being black, 1 being white and 0.5 being the maximum amount of color. In other words as luminance goes from 0.0 to 0.5 the color will go from black to hue. From 0.5 to 1.0 the color will go from hue to white.

We also passed side: THREE.DoubleSide to the material. This tells three to draw both sides of the triangles that make up a shape. For a solid shape like a sphere or a cube there's usually no reason to draw the back sides of triangles as they all face inside the shape. In our case though we are drawing a few things like the PlaneGeometry and the ShapeGeometry which are 2 dimensional and so have no inside. Without setting side: THREE.DoubleSide they would disappear when looking at their back sides.

I should note that it's faster to draw when not setting side: THREE.DoubleSide so ideally we'd set it only on the materials that really need it but in this case we are not drawing too much so there isn't much reason to worry about it.

Let's make a function, addSolidGeometry, that we pass a geometry and it creates a random colored material via createMaterial and adds it to the scene via addObject.

Now we can use this for the majority of the primitives we create. For example creating a box

If you look in the code below you'll see a similar section for each type of geometry.

javascript

```
const scene = new THREE.Scene();
+scene.background = new THREE.Color(0xAAAAAA);
```

javascript

```
-const fov = 75;
+const fov = 40;
const aspect = 2;  // the canvas default
const near = 0.1;
-const far = 5;
+const far = 1000;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
-camera.position.z = 2;
+camera.position.z = 120;
```

javascript

```
const objects = [];
const spread = 15;

function addObject(x, y, obj) {
  obj.position.x = x * spread;
  obj.position.y = y * spread;

  scene.add(obj);
  objects.push(obj);
}
```

javascript

```
function createMaterial() {
  const material = new THREE.MeshPhongMaterial({
    side: THREE.DoubleSide,
  });

  const hue = Math.random();
  const saturation = 1;
  const luminance = .5;
  material.color.setHSL(hue, saturation, luminance);

  return material;
}
```

javascript

```
function addSolidGeometry(x, y, geometry) {
  const mesh = new THREE.Mesh(geometry, createMaterial());
  addObject(x, y, mesh);
}
```

javascript

```
{
  const width = 8;
  const height = 8;
  const depth = 8;
  addSolidGeometry(-2, -2, new THREE.BoxGeometry(width, height, depth));
}
```

TextGeometry and Font Loading

There are a couple of notable exceptions to the pattern above. The biggest is probably the TextGeometry. It needs to load 3D font data before it can generate a mesh for the text. That data loads asynchronously so we need to wait for it to load before trying to create the geometry. By promisifiying font loading we can make it mush easier. We

create a FontLoader and then a function loadFont that returns a promise that on resolve will give us the font. We then create an async function called doit and load the font using await. And finally create the geometry and call addObject to add it the scene.

There's one other difference. We want to spin the text around its center but by default three.js creates the text such that its center of rotation is on the left edge. To work around this we can ask three.js to compute the bounding box of the geometry. We can then call the getCenter method of the bounding box and pass it our mesh's position object. getCenter copies the center of the box into the position. It also returns the position object so we can call multiplyScalar(-1) to position the entire object such that its center of rotation is at the center of the object.

If we then just called addSolidGeometry like with previous examples it would set the position again which is no good. So, in this case we create an Object3D which is the standard node for the three.js scene graph. Mesh is inherited from Object3D as well. We'll cover how the scene graph works in another article. For now it's enough to know that like DOM nodes, children are drawn relative to their parent. By making an Object3D and making our mesh a child of that we can position the Object3D wherever we want and still keep the center offset we set earlier.

If we didn't do this the text would spin off center.

```
javascript
```

```
{
  const loader = new FontLoader();
  // promisify font loading
  function loadFont(url) {
    return new Promise((resolve, reject) => {
      loader.load(url, resolve, undefined, reject);
    });
  }

  async function doit() {
    const font = await loadFont('resources/threejs/fonts/helvetiker_regular.typefac
    const geometry = new TextGeometry('three.js', {
      font: font,
      size: 3.0,
      depth: .2,
      curveSegments: 12,
      bevelEnabled: true,
      bevelThickness: 0.15,
      bevelSize: .3,
      bevelSegments: 5,
    });
    const mesh = new THREE.Mesh(geometry, createMaterial());
    geometry.computeBoundingBox();
    geometry.boundingBox.getCenter(mesh.position).multiplyScalar(-1);

    const parent = new THREE.Object3D();
    parent.add(mesh);

    addObject(-1, -1, parent);
  }
  doit();
}
```

Line Geometry Helpers

The other exceptions are the 2 line based examples for EdgesGeometry and WireframeGeometry. Instead of calling addSolidGeometry they call addLineGeometry which looks like this

It creates a black LineBasicMaterial and then creates a LineSegments object which is a wrapper for Mesh that helps three know you're rendering line segments (2 points per segment).

Each of the primitives has several parameters you can pass on creation and it's best to look in the documentation for all of them rather than repeat them here. You can also click the links above next to each shape to take you directly to the docs for that shape.

```
javascript
```

```
function addLineGeometry(x, y, geometry) {
  const material = new THREE.LineBasicMaterial({color: 0x000000});
  const mesh = new THREE.LineSegments(geometry, material);
  addObject(x, y, mesh);
}
```

Points and PointsMaterial

There is one other pair of classes that doesn't really fit the patterns above. Those are the PointsMaterial and the Points class. Points is like LineSegments above in that it takes a a BufferGeometry but draws points at each vertex instead of lines. To use it you also need to pass it a PointsMaterial which take a size for how large to make the points.

You can turn off sizeAttenuation by setting it to false if you want the points to be the same size regardless of their distance from the camera.

javascript

```
const radius = 7;
const widthSegments = 12;
const heightSegments = 8;
const geometry = new THREE.SphereGeometry(radius, widthSegments, heightSegments);
const material = new THREE.PointsMaterial({
    color: 'red',
    size: 0.2,     // in world units
});
const points = new THREE.Points(geometry, material);
scene.add(points);
```

javascript

```
const material = new THREE.PointsMaterial({
    color: 'red',
+    sizeAttenuation: false,
+    size: 3,       // in pixels
-    size: 0.2,     // in world units
});
...
```

Subdivisions

One other thing that's important to cover is that almost all shapes have various settings for how much to subdivide them. A good example might be the sphere geometries. Spheres take parameters for how many divisions to make around and how many top to bottom. For example

The first sphere has 5 segments around and 3 high which is 15 segments or 30 triangles. The second sphere has 24 segments by 10. That's 240 segments or 480 triangles. The last one has 50 by 50 which is 2500 segments or 5000 triangles.

It's up to you to decide how many subdivisions you need. It might look like you need a high number of segments but remove the lines and the flat shading and we get this

It's now not so clear that the one on the right with 5000 triangles is entirely better than the one in the middle with only 480. If you're only drawing a few spheres, like say a single globe for a map of the earth, then a single 10000 triangle sphere is not a bad choice. If on the other hand you're trying to draw 1000 spheres then 1000 spheres times 10000 triangles each is 10 million triangles. To animate smoothly you need the browser to draw at 60 frames per second so you'd be asking the browser to draw 600 million triangles per second. That's a lot of computing.

Sometimes it's easy to choose. For example you can also choose to subdivide a plane.

The plane on the left is 2 triangles. The plane on the right is 200 triangles. Unlike the sphere there is really no trade off in quality for most use cases of a plane. You'd most likely only subdivide a plane if you expected to want to modify or warp it in some way. A box is similar.

So, choose whatever is appropriate for your situation. The less subdivisions you choose the more likely things will run smoothly and the less memory they'll take. You'll have to decide for yourself what the correct tradeoff is for your particular situation.

If none of the shapes above fit your use case you can load geometry for example from a .obj file or a .gltf file. You can also create your own custom BufferGeometry.

Animation System

REFERENCE

Source: https://threejs.org/manual/en/animation-system.html

Extraction fallback content. [stop]

Animation System

Animation System

Overview

```
Within the three.js animation system you can animate various proper
the bones of a skinned and rigged model, morph targets, different m
(colors, opacity, booleans), visibility and transforms. The animate
faded out, crossfaded and warped. The weight and time scales of dif
animations on the same object as well as on different objects can b
independently. Various animations on the same and on different obje
synchronized.



To achieve all this in one homogeneous system, the three.js animati
[has completely changed in 2015](https://github.com/mrdoob/three.js
(beware of outdated information!), and it has now an architecture s
Unity/Unreal Engine 4. This page gives a short overview of the main
system and how they work together.
```

Animation Clips

```
If you have successfully imported an animated 3D object (it doesn't
bones or morph targets or both) – for example exporting it from Ble
[glTF Blender exporter](https://github.com/KhronosGroup/glTF-Blende
loading it into a three.js scene using `GLTFLoader` – one of the re
should be an array named "animations", containing the animation cli
for this model (see a list of possible loaders below).



Each `AnimationClip` usually holds the data for a certain activity
mesh is a character, for example, there may be one AnimationClip fo
for a jump, a third for sidestepping and so on.
```

Keyframe Tracks

```
Inside of such an `AnimationClip` the data for each animated proper
separate `KeyframeTrack`. Assuming a character object has a skeleto
one keyframe track could store the data for the position changes of
over time, a different track the data for the rotation changes of t
the track position, rotation or scaling of another bone, and so on.
that an AnimationClip can be composed of lots of such tracks.



Assuming the model has morph targets (for example one morph
target showing a friendly face and another showing an angry face),
information as to how the influence of a certain morph target chang
of the clip.
```

Animation Mixer

```
The stored data forms only the basis for the animations - actual pl
the `AnimationMixer`. You can imagine this not only as a player for
as a simulation of a hardware like a real mixer console, which can
simultaneously, blending and merging them.
```

Animation Actions

```
The `AnimationMixer` itself has only very few (general) properties
can be controlled by the animation actions. By configuring an
`AnimationAction` you can determine when a certain `AnimationClip`
or stopped on one of the mixers, if and how often the clip has to b
shall be performed with a fade or a time scaling, and some addition
or synchronizing.
```

Animation Object Groups

```
If you want a group of objects to receive a shared animation state,
`AnimationObjectGroup`.
```

Supported Formats and Loaders

```
Note that not all model formats include animation (OBJ notably does
three.js loaders support `AnimationClip` sequences. Several that  *
support this animation type:
```

- THREE.ObjectLoader
- THREE.BVHLoader
- THREE.ColladaLoader
- THREE.FBXLoader
- THREE.GLTFLoader

> ```
> Note that 3ds max and Maya currently can't export multiple animations (
> on the same timeline) directly to a single file.
> ```

Example

```
js
```

```
let mesh;

// Create an AnimationMixer, and get the list of AnimationClip instances
const mixer = new THREE.AnimationMixer( mesh );
const clips = mesh.animations;

// Update the mixer on each frame
function update () {
  mixer.update( deltaSeconds );
}

// Play a specific animation
const clip = THREE.AnimationClip.findByName( clips, 'dance' );
const action = mixer.clipAction( clip );
action.play();

// Play all animations
clips.forEach( function ( clip ) {
  mixer.clipAction( clip ).play();
} );
```

Core Features

Color Management

GUIDE

Source: https://threejs.org/manual/en/color-management.html

An overview of color space concepts in three.js, covering color primaries, white points, transfer functions, and how input, working, and output color spaces are managed during rendering. Includes guidance on working with THREE.Color instances and common pitfalls.

What is a color space?

Every color space is a collection of several design decisions, chosen together to support a large range of colors while satisfying technical constraints related to precision and display technologies. When creating a 3D asset, or assembling 3D assets together into a scene, it is important to know what these properties are, and how the properties of one color space relate to other color spaces in the scene.

sRGB colors and white point (D65) displayed in the reference CIE 1931 chromaticity diagram. Colored region represents a 2D projection of the sRGB gamut, which is a 3D volume.

- Color primaries: Primary colors (e.g. red, green, blue) are not absolutes; they are selected from the visible spectrum based on constraints of limited precision and capabilities of available display devices. Colors are expressed as a ratio of the primary colors.

- White point: Most color spaces are engineered such that an equally weighted sum of primaries R = G = B will appear to be without color, or "achromatic". The appearance of achromatic values (like white or grey) depend on human perception, which in turn depends heavily on the context of the observer. A color space specifies its "white point" to balance these needs. The white point defined by the sRGB color space is D65.

- Transfer functions: After choosing the color gamut and a color model, we still need to define mappings ("transfer functions") of numerical values to/from the color space. Does r = 0.5 represent 50% less physical illumination than r = 1.0? Or 50% less bright, as perceived by an average human eye? These are different things, and that difference can be represented as a mathematical function. Transfer functions may be linear or nonlinear, depending on the objectives of the color space. sRGB defines nonlinear transfer functions. Those functions are sometimes approximated as gamma functions, but the term "gamma" is ambiguous and should be avoided in this context.

These three parameters — color primaries, white point, and transfer functions — define a color space, with each chosen for particular goals. Having defined the parameters, a few additional terms are helpful:

- Color model: Syntax for numerically identifying colors within chosen the color gamut — a coordinate system for colors. In three.js we're mainly concerned with the RGB color model, having three coordinates r, g, b $\in$ [0,1] ("closed domain") or r, g, b $\in$ [0,∞] ("open domain") each representing a fraction of a primary color. Other color models (HSL, Lab, LCH) are commonly used for artistic control.

- Color gamut: Once color primaries and a white point have been chosen, these represent a volume within the visible spectrum (a "gamut"). Colors not within this volume ("out of gamut") cannot be expressed by

closed domain [0,1] RGB values. In the open domain [0,∞], the gamut is technically infinite.

Consider two very common color spaces: SRGBColorSpace ("sRGB") and LinearSRGBColorSpace ("Linear-sRGB"). Both use the same primaries and white point, and therefore have the same color gamut. Both use the RGB color model. They differ only in the transfer functions — Linear-sRGB is linear with respect to physical light intensity. sRGB uses the nonlinear sRGB transfer functions, and more closely resembles the way that the human eye perceives light and the responsiveness of common display devices.

That difference is important. Lighting calculations and other rendering operations must generally occur in a linear color space. However, a linear colors are less efficient to store in an image or framebuffer, and do not look correct when viewed by a human observer. As a result, input textures and the final rendered image will generally use the nonlinear sRGB color space.

NOTICE: While some modern displays support wider gamuts like Display-P3, the web platform's graphics APIs largely rely on sRGB. Applications using three.js today will typically use only the sRGB and Linear-sRGB color spaces.

Roles of color spaces

Linear workflows — required for modern rendering methods — generally involve more than one color space, each assigned to a particular role. Linear and nonlinear color spaces are appropriate for different roles, explained below.

Input color space

Colors supplied to three.js — from color pickers, textures, 3D models, and other sources — each have an associated color space. Those not already in the Linear-sRGB working color space must be converted, and textures be given the correct texture.colorSpace assignment. Certain conversions (for hexadecimal and CSS colors in sRGB) can be made automatically if the THREE.ColorManagement API is enabled before initializing colors:

THREE.ColorManagement is enabled by default.

- Materials, lights, and shaders: Colors in materials, lights, and shaders store RGB components in the Linear-sRGB working color space.

- Vertex colors: BufferAttribute store RGB components in the Linear-sRGB working color space.
- Color textures: PNG or JPEG Texture containing color information (like .map or .emissiveMap) use the closed domain sRGB color space, and must be annotated with texture.colorSpace = SRGBColorSpace. Formats like OpenEXR (sometimes used for .envMap or .lightMap) use the Linear-sRGB color space indicated with texture.colorSpace = LinearSRGBColorSpace, and may contain values in the open domain [0,∞].
- Non-color textures: Textures that do not store color information (like .normalMap or .roughnessMap) do not have an associated color space, and generally use the (default) texture annotation of texture.colorSpace = NoColorSpace. In rare cases, non-color data may be represented with other nonlinear encodings for technical reasons.

WARNING: Many formats for 3D models do not correctly or consistently define color space information. While three.js attempts to handle most cases, problems are common with older file formats. For best results, use glTF 2.0 (GLTFLoader) and test 3D models in online viewers early to confirm the asset itself is correct.

```javascript
THREE.ColorManagement.enabled = true;
```

Working color space

Rendering, interpolation, and many other operations must be performed in an open domain linear working color space, in which RGB components are proportional to physical illumination. In three.js, the working color space is Linear-sRGB.

Output color space

Output to a display device, image, or video may involve conversion from the open domain Linear-sRGB working color space to another color space. The conversion is defined by (WebGLRenderer.outputColorSpace). When using post-processing, this requires OutputPass.

- Display: Colors written to a WebGL canvas for display should be in the sRGB color space.

- Image: Colors written to an image should use the color space appropriate for the format and usage. Fully-rendered images written to PNG or JPEG textures generally use the sRGB color space. Images containing emission, light maps, or other data not confined to the [0,1] range will generally use the open domain Linear-sRGB color space, and a compatible image format like OpenEXR.

WARNING: Render targets may use either sRGB or Linear-sRGB. sRGB makes better use of limited precision. In the closed domain, 8 bits often suffice for sRGB whereas ≥12 bits (half float) may be required for Linear-sRGB. If later pipeline stages require Linear-sRGB input, the additional conversions may have a small performance cost.

Custom materials based on ShaderMaterial and RawShaderMaterial have to implement their own output color space conversion. For instances of ShaderMaterial, adding the colorspace_fragment shader chunk to the fragment shader's main() function should be sufficient.

Working with THREE.Color instances

Methods reading or modifying Color instances assume data is already in the three.js working color space, Linear-sRGB. RGB and HSL components are direct representations of data stored by the Color instance, and are never converted implicitly. Color data may be explicitly converted with .convertLinearToSRGB() or .convertSRGBToLinear().

With ColorManagement.enabled = true set (recommended), certain conversions are made automatically. Because hexadecimal and CSS colors are generally sRGB, Color methods will automatically convert these inputs from sRGB to Linear-sRGB in setters, or convert from Linear-sRGB to sRGB when returning hexadecimal or CSS output from getters.

javascript

```javascript
// RGB components (no change).
color.r = color.g = color.b = 0.5;
console.log( color.r ); // → 0.5

// Manual conversion.
color.r = 0.5;
color.convertSRGBToLinear();
console.log( color.r ); // → 0.214041140
```

javascript

```
// Hexadecimal conversion.
color.setHex( 0x808080 );
console.log( color.r ); // → 0.214041140
console.log( color.getHex() ); // → 0x808080

// CSS conversion.
color.setStyle( 'rgb( 0.5, 0.5, 0.5 )' );
console.log( color.r ); // → 0.214041140

// Override conversion with 'colorSpace' argument.
color.setHex( 0x808080, LinearSRGBColorSpace );
console.log( color.r ); // → 0.5
console.log( color.getHex( LinearSRGBColorSpace ) ); // → 0x808080
console.log( color.getHex( SRGBColorSpace ) ); // → 0xBCBCBC
```

Common mistakes

When an individual color or texture is misconfigured, it will appear darker or lighter than expected. When the renderer's output color space is misconfigured, the entire scene may appear darker (e.g. missing conversion to sRGB) or lighter (e.g. a double conversion to sRGB with post-processing). In each case the problem may not be uniform, and simply increasing/decreasing lighting does not solve it.

A more subtle issue appears when both the input color spaces and the output color spaces are incorrect — the overall brightness levels may be fine, but colors may change unexpectedly under different lighting, or shading may appear more blown-out and less soft than intended. These two wrongs do not make a right, and it's important that the working color space be linear ("scene referred") and the output color space be nonlinear ("display referred").

Further reading

- GPU Gems 3: The Importance of Being Linear, by Larry Gritz and Eugene d'Eon
- What every coder should know about gamma, by John Novak
- The Hitchhiker's Guide to Digital Color, by Troy Sobotka
- Color Management, Blender

Materials

GUIDE

Source: https://threejs.org/manual/en/materials.html

An overview of the materials available in three.js, covering how to set material properties, the different material types (Basic, Lambert, Phong, Toon, Standard,

Physical, and special-use materials), and common properties shared across materials.

Introduction to Materials

Three.js provides several types of materials. They define how objects will appear in the scene. Which materials you use really depends on what you're trying to accomplish.

This article is part of a series of articles about three.js. The first article is three.js fundamentals. If you haven't read that yet and you're new to three.js you might want to consider starting there.

Setting Material Properties

There are 2 ways to set most material properties. One at creation time which we've seen before.

The other is after creation.

javascript

```
const material = new THREE.MeshPhongMaterial({
  color: 0xFF0000,    // red (can also use a CSS color string here)
  flatShading: true,
});
```

javascript

```
const material = new THREE.MeshPhongMaterial();
material.color.setHSL(0, 1, .5);  // red
material.flatShading = true;
```

THREE.Color Properties

Note that properties of type THREE.Color have multiple ways to be set.

javascript

```
material.color.set(0x00FFFF);    // same as CSS's #RRGGBB style
material.color.set(cssString);   // any CSS color, eg 'purple', '#F32',
                                 // 'rgb(255, 127, 64)',
                                 // 'hsl(180, 50%, 25%)'
material.color.set(someColor)    // some other THREE.Color
material.color.setHSL(h, s, l)   // where h, s, and l are 0 to 1
material.color.setRGB(r, g, b)   // where r, g, and b are 0 to 1
```

Color Strings at Creation Time

At creation time you can pass either a hex number or a CSS string.

```javascript
const m1 = new THREE.MeshBasicMaterial({color: 0xFF0000});         // red
const m2 = new THREE.MeshBasicMaterial({color: 'red'});            // red
const m3 = new THREE.MeshBasicMaterial({color: '#F00'});           // red
const m4 = new THREE.MeshBasicMaterial({color: 'rgb(255,0,0)'});   // red
const m5 = new THREE.MeshBasicMaterial({color: 'hsl(0,100%,50%)'}); // red
```

Basic, Lambert, and Phong Materials

The MeshBasicMaterial is not affected by lights. The MeshLambertMaterial computes lighting only at the vertices vs the MeshPhongMaterial which computes lighting at every pixel. The MeshPhongMaterial also supports specular highlights.

Why have all 3 when MeshPhongMaterial can do the same things as MeshBasicMaterial and MeshLambertMaterial? The reason is the more sophisticated material takes more GPU power to draw. On a slower GPU like say a mobile phone you might want to reduce the GPU power needed to draw your scene by using one of the less complex materials. It also follows that if you don't need the extra features then use the simplest material. If you don't need the lighting and the specular highlight then use the MeshBasicMaterial.

Shininess

The shininess setting of the MeshPhongMaterial determines the shininess of the specular highlight. It defaults to 30.

Note that setting the emissive property to a color on either a MeshLambertMaterial or a MeshPhongMaterial and setting the color to black (and shininess to 0 for phong) ends up looking just like the MeshBasicMaterial.

MeshToonMaterial

The MeshToonMaterial is similar to the MeshPhongMaterial with one big difference. Rather than shading smoothly it uses a gradient map (an X by 1 texture) to decide how to shade. The default uses a gradient map that is 70% brightness for the first 70% and 100% after but you can supply your own gradient map. This ends up giving a 2 tone look that looks like a cartoon.

Physically Based Rendering (PBR) Materials

Next up there are 2 physically based rendering materials. Physically Based Rendering is often abbreviated PBR.

The materials above use simple math to make materials that look 3D but they aren't what actually happens in real world. The 2 PBR materials use much more complex math to come close to what actually happens in the real world.

MeshStandardMaterial

The first one is MeshStandardMaterial. The biggest difference between MeshPhongMaterial and MeshStandardMaterial is it uses different parameters. MeshPhongMaterial had a shininess setting. MeshStandardMaterial has 2 settings roughness and metalness.

At a basic level roughness is the opposite of shininess. Something that has a high roughness, like a baseball doesn't have hard reflections whereas something that's not rough, like a billiard ball, is very shiny. Roughness goes from 0 to 1.

The other setting, metalness, says how metal the material is. Metals behave differently than non-metals. 0 for non-metal and 1 for metal.

MeshPhysicalMaterial

The MeshPhysicalMaterial is same as the MeshStandardMaterial but it adds a clearcoat parameter that goes from 0 to 1 for how much to apply a clearcoat gloss layer and a clearCoatRoughness parameter that specifies how rough the gloss layer is.

Material Performance Progression

The various standard materials progress from fastest to slowest MeshBasicMaterial ➡ MeshLambertMaterial ➡ MeshPhongMaterial ➡ MeshStandardMaterial ➡ MeshPhysicalMaterial. The slower materials can make more realistic looking scenes but you might need to design your code to use the faster materials on low powered or mobile machines.

ShadowMaterial

There are 3 materials that have special uses. ShadowMaterial is used to get the data created from shadows. We haven't covered shadows yet. When we do we'll use this material to take a peek at what's happening behind the scenes.

MeshDepthMaterial

The MeshDepthMaterial renders the depth of each pixel where pixels at negative near of the camera are 0 and negative far are 1. Certain special effects can use this data which we'll get into at another time.

MeshNormalMaterial

The MeshNormalMaterial will show you the normals of geometry. Normals are the direction a particular triangle or pixel faces. MeshNormalMaterial draws the view space normals (the normals relative to the camera). x is red, y is green, and z is blue so things facing to the right will be pink, to the left will be aqua, up will be light green, down will be purple, and toward the screen will be lavender.

ShaderMaterial and RawShaderMaterial

ShaderMaterial is for making custom materials using the three.js shader system. RawShaderMaterial is for making entirely custom shaders with no help from three.js. Both of these topics are large and will be covered later.

Common Material Properties

Most materials share a bunch of settings all defined by Material. See the docs for all of them but let's go over two of the most commonly used properties.

flatShading: whether or not the object looks faceted or smooth. default = false.

side: which sides of triangles to show. The default is THREE.FrontSide. Other options are THREE.BackSide and THREE.DoubleSide (both sides). Most 3D objects drawn in three are probably opaque solids so the back sides (the sides facing inside the solid) do not need to be drawn. The most common reason to set side is for planes or other non-solid objects where it is common to see the back sides of triangles.

material.needsUpdate

This topic rarely affects most three.js apps but just as an FYI... Three.js applies material settings when a material is used where "used" means "something is rendered that uses the material". Some material settings are only applied once as changing them requires lots of work by three.js. In those cases you need to set material.needsUpdate = true to tell three.js to apply your material changes. The most common settings that require you to set needsUpdate if you change the settings after using the material are:

- flatShading

- adding or removing a texture

Changing a texture is ok, but if want to switch from using no texture to using a texture or from using a texture to using no texture then you need to set needsUpdate = true.

In the case of going from texture to no-texture it is often just better to use a 1x1 pixel white texture.

As mentioned above most apps never run into these issues. Most apps do not switch between flat shaded and non flat shaded. Most apps also either use textures or a solid color for a given material, they rarely switch from using one to using the other.

Textures

GUIDE

Source: https://threejs.org/manual/en/textures.html

A comprehensive guide to textures in Three.js covering how to create, load, and apply textures to materials, including memory usage considerations, filtering and mipmaps, and texture wrapping/repeating/offsetting/rotating settings.

Hello Texture

Textures are *generally* images that are most often created in some 3rd party program like Photoshop or GIMP. For example let's put this image on cube.

We'll modify one of our first samples. All we need to do is create a TextureLoader. Call its load method with the URL of an image and set the material's `map` property to the result instead of setting its `color`.

Note that we're using MeshBasicMaterial so no need for any lights.

```javascript
const loader = new THREE.TextureLoader();
const texture = loader.load( 'resources/images/wall.jpg' );
texture.colorSpace = THREE.SRGBColorSpace;

const material = new THREE.MeshBasicMaterial({
  map: texture,
});
```

6 Textures, a different one on each face of a cube

How about 6 textures, one on each face of a cube?

We just make 6 materials and pass them as an array when we create the Mesh

It works!

It should be noted though that not all geometry types supports multiple materials. BoxGeometry can use 6 materials one for each face. ConeGeometry can use 2 materials, one for the bottom and one for the cone. CylinderGeometry can use 3 materials, bottom, top, and side. For other cases you will need to build or load custom geometry and/or modify texture coordinates.

It's far more common in other 3D engines and far more performant to use a Texture Atlas if you want to allow multiple images on a single geometry. A Texture atlas is where you put multiple images in a single texture and then use texture coordinates on the vertices of your geometry to select which parts of a texture are used on each triangle in your geometry.

What are texture coordinates? They are data added to each vertex of a piece of geometry that specify what part of the texture corresponds to that specific vertex. We'll go over them when we start building custom geometry.

```javascript
const loader = new THREE.TextureLoader();
const materials = [
  new THREE.MeshBasicMaterial({map: loadColorTexture('resources/images/flower-1.jpg
  new THREE.MeshBasicMaterial({map: loadColorTexture('resources/images/flower-2.jpg
  new THREE.MeshBasicMaterial({map: loadColorTexture('resources/images/flower-3.jpg
  new THREE.MeshBasicMaterial({map: loadColorTexture('resources/images/flower-4.jpg
  new THREE.MeshBasicMaterial({map: loadColorTexture('resources/images/flower-5.jpg
  new THREE.MeshBasicMaterial({map: loadColorTexture('resources/images/flower-6.jpg
];
const cube = new THREE.Mesh(geometry, materials);

function loadColorTexture( path ) {
  const texture = loader.load( path );
  texture.colorSpace = THREE.SRGBColorSpace;
  return texture;
}
```

Loading Textures

The Easy Way

Most of the code on this site uses the easiest method of loading textures. We create a TextureLoader and then call its load method. This returns a Texture object.

It's important to note that using this method our texture will be transparent until the image is loaded asynchronously by three.js at which point it will update the texture with the downloaded image.

This has the big advantage that we don't have to wait for the texture to load and our page will start rendering immediately. That's probably okay for a great many use cases but if we want we can ask three.js to tell us when the texture has finished downloading.

```javascript
const texture = loader.load('resources/images/flower-1.jpg');
```

Waiting for a texture to load

To wait for a texture to load the `load` method of the texture loader takes a callback that will be called when the texture has finished loading. Going back to our top example we can wait for the texture to load before creating our Mesh and adding it to scene like this

Unless you clear your browser's cache and have a slow connection you're unlikely to see the any difference but rest assured it is waiting for the texture to load.

```javascript
const loader = new THREE.TextureLoader();
loader.load('resources/images/wall.jpg', (texture) => {
  texture.colorSpace = THREE.SRGBColorSpace;
  const material = new THREE.MeshBasicMaterial({
    map: texture,
  });
  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);
  cubes.push(cube);  // add to our list of cubes to rotate
});
```

Waiting for multiple textures to load

To wait until all textures have loaded you can use a LoadingManager. Create one and pass it to the TextureLoader then set its onLoad property to a callback.

The LoadingManager also has an onProgress property we can set to another callback to show a progress indicator.

First we'll add a progress bar in HTML

and the CSS for it

Then in the code we'll update the scale of the `progressbar` in our `onProgress` callback. It gets called with the URL of the last item loaded, the number of items loaded so far, and the total number of items loaded.

Unless you clear your cache and have a slow connection you might not see the loading bar.

javascript

```
const loadManager = new THREE.LoadingManager();
const loader = new THREE.TextureLoader(loadManager);

const materials = [
  new THREE.MeshBasicMaterial({map: loader.load('resources/images/flower-1.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('resources/images/flower-2.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('resources/images/flower-3.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('resources/images/flower-4.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('resources/images/flower-5.jpg')}),
  new THREE.MeshBasicMaterial({map: loader.load('resources/images/flower-6.jpg')}),
];

loadManager.onLoad = () => {
  const cube = new THREE.Mesh(geometry, materials);
  scene.add(cube);
  cubes.push(cube);  // add to our list of cubes to rotate
};
```

html

```
<body>
  <canvas id="c"></canvas>
  <div id="loading">
    <div class="progress"><div class="progressbar"></div></div>
  </div>
</body>
```

css

```
#loading {
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    display: flex;
    justify-content: center;
    align-items: center;
}
#loading .progress {
    margin: 1.5em;
    border: 1px solid white;
    width: 50vw;
}
#loading .progressbar {
    margin: 2px;
    background: white;
    height: 1em;
    transform-origin: top left;
    transform: scaleX(0);
}
```

javascript

```
const loadingElem = document.querySelector('#loading');
const progressBarElem = loadingElem.querySelector('.progressbar');

loadManager.onLoad = () => {
  loadingElem.style.display = 'none';
  const cube = new THREE.Mesh(geometry, materials);
  scene.add(cube);
  cubes.push(cube);  // add to our list of cubes to rotate
};

loadManager.onProgress = (urlOfLastItemLoaded, itemsLoaded, itemsTotal) => {
  const progress = itemsLoaded / itemsTotal;
  progressBarElem.style.transform = scaleX(${progress});
};
```

Loading textures from other origins

To use images from other servers those servers need to send the correct headers. If they don't you cannot use the images in three.js and will get an error. If you run the server providing the images make sure it sends the correct headers. If you don't control the server hosting the images and it does not send the permission headers then you can't use the images from that server.

For example imgur, flickr, and github all send headers allowing you to use images hosted on their servers in three.js. Most other websites do not.

Memory Usage

Textures are often the part of a three.js app that use the most memory. It's important to understand that *in general*, textures take `width * height * 4 * 1.33` bytes of memory.

Notice that says nothing about compression. I can make a .jpg image and set its compression super high. For example let's say I was making a scene of a house. Inside the house there is a table and I decide to put this wood texture on the top surface of the table

That image is only 157k so it will download relatively quickly but it is actually 3024 x 3761 pixels in size. Following the equation above that's

3024 * 3761 * 4 * 1.33 = 60505764.5

That image will take 60 MEG OF MEMORY! in three.js. A few textures like that and you'll be out of memory.

I bring this up because it's important to know that using textures has a hidden cost. In order for three.js to use the texture it has to hand it off to the GPU and the GPU *in general* requires the texture data to be uncompressed.

The moral of the story is make your textures small in dimensions not just small in file size. Small in file size = fast to download. Small in dimensions = takes less memory. How small should you make them? As small as you can and still look as good as you need them to look.

JPG vs PNG

This is pretty much the same as regular HTML in that JPGs have lossy compression, PNGs have lossless compression so PNGs are generally slower to download. But, PNGs support transparency. PNGs are also probably the appropriate format for non-image data like normal maps, and other kinds of non-image maps which we'll go over later.

It's important to remember that a JPG doesn't use less memory than a PNG in WebGL. See above.

Filtering and Mips

Let's apply this 16x16 texture

To a cube

Let's draw that cube really small

Hmmm, I guess that's hard to see. Let's magnify that tiny cube

How does the GPU know which colors to make each pixel it's drawing for the tiny cube? What if the cube was so small that it's just 1 or 2 pixels?

This is what filtering is about.

If it was Photoshop, Photoshop would average nearly all the pixels together to figure out what color to make those 1 or 2 pixels. That would be a very slow operation. GPUs solve this issue using mipmaps.

Mips are copies of the texture, each one half as wide and half as tall as the previous mip where the pixels have been blended to make the next smaller mip. Mips are created until we get all the way to a 1x1 pixel mip.

Now, when the cube is drawn so small that it's only 1 or 2 pixels large the GPU can choose to use just the smallest or next to smallest mip level to decide what color to make the tiny cube.

In three.js you can choose what happens both when the texture is drawn larger than its original size and what happens when it's drawn smaller than its original size.

For setting the filter when the texture is drawn larger than its original size you set `texture.magFilter` property to either `THREE.NearestFilter` or `THREE.LinearFilter`. `NearestFilter` means just pick the closet single pixel from the original texture. With a low resolution texture this gives you a very pixelated look like Minecraft.

`LinearFilter` means choose the 4 pixels from the texture that are closest to the where we should be choosing a color from and blend them in the appropriate proportions relative to how far away the actual point is from each of the 4 pixels.

Nearest vs Linear comparison shown.

For setting the filter when the texture is drawn smaller than its original size you set the `texture.minFilter` property to one of 6 values:

- `THREE.NearestFilter` — same as above, choose the closest pixel in the texture
- `THREE.LinearFilter` — same as above, choose 4 pixels from the texture and blend them
- `THREE.NearestMipmapNearestFilter` — choose the appropriate mip then choose one pixel

- `THREE.NearestMipmapLinearFilter` — choose 2 mips, choose one pixel from each, blend the 2 pixels
- `THREE.LinearMipmapNearestFilter` — chose the appropriate mip then choose 4 pixels and blend them
- `THREE.LinearMipmapLinearFilter` — choose 2 mips, choose 4 pixels from each and blend all 8 into 1 pixel

Here's an example showing all 6 settings

One thing to notice is the top left and top middle using `NearestFilter` and `LinearFilter` don't use the mips. Because of that they flicker in the distance because the GPU is picking pixels from the original texture. On the left just one pixel is chosen and in the middle 4 are chosen and blended but it's not enough come up with a good representative color. The other 4 strips do better with the bottom right, `LinearMipmapLinearFilter` being best.

If you click the picture above it will toggle between the texture we've been using above and a texture where every mip level is a different color.

This makes it more clear what is happening. You can see in the top left and top middle the first mip is used all the way into the distance. The top right and bottom middle you can clearly see where a different mip is used.

Switching back to the original texture you can see the bottom right is the smoothest, highest quality. You might ask why not always use that mode. The most obvious reason is sometimes you want things to be pixelated for a retro look or some other reason. The next most common reason is that reading 8 pixels and blending them is slower than reading 1 pixel and blending. While it's unlikely that a single texture is going to be the difference between fast and slow as we progress further into these articles we'll eventually have materials that use 4 or 5 textures all at once. 4 textures * 8 pixels per texture is looking up 32 pixels for ever pixel rendered. This can be especially important to consider on mobile devices.

Repeating, offseting, rotating, wrapping a texture

Textures have settings for repeating, offseting, and rotating a texture.

By default textures in three.js do not repeat. To set whether or not a texture repeats there are 2 properties, `wrapS` for horizontal wrapping and `wrapT` for vertical wrapping.

They can be set to one of:

- `THREE.ClampToEdgeWrapping` — the last pixel on each edge is repeated forever
- `THREE.RepeatWrapping` — the texture is repeated
- `THREE.MirroredRepeatWrapping` — the texture is mirrored and repeated

For example to turn on wrapping in both directions:

Repeating is set with the `repeat` property.

Offseting the texture can be done by setting the `offset` property. Textures are offset with units where 1 unit = 1 texture size. On other words 0 = no offset and 1 = offset one full texture amount.

Rotating the texture can be set by setting the `rotation` property in radians as well as the `center` property for choosing the center of rotation. It defaults to 0,0 which rotates from the bottom left corner. Like offset these units are in texture size so setting them to `.5, .5` would rotate around the center of the texture.

Let's modify the top sample above to play with these values

First we'll keep a reference to the texture so we can manipulate it

Then we'll use lil-gui again to provide a simple interface.

As we did in previous lil-gui examples we'll use a simple class to give lil-gui an object that it can manipulate in degrees but that will set a property in radians.

We also need a class that will convert from a string like `"123"` into a number like `123` since three.js requires numbers for enum settings like `wrapS` and `wrapT` but lil-gui only uses strings for enums.

Using those classes we can setup a simple GUI for the settings above

The last thing to note about the example is that if you change `wrapS` or `wrapT` on the texture you must also set `texture.needsUpdate` so three.js knows to apply those settings. The other settings are automatically applied.

This is only one step into the topic of textures. At some point we'll go over texture coordinates as well as 9 other types of textures that can be applied to materials.

For now let's move on to lights.

javascript

```
someTexture.wrapS = THREE.RepeatWrapping;
someTexture.wrapT = THREE.RepeatWrapping;
```

javascript

```
const timesToRepeatHorizontally = 4;
const timesToRepeatVertically = 2;
someTexture.repeat.set(timesToRepeatHorizontally, timesToRepeatVertically);
```

javascript

```
const xOffset = .5;   // offset by half the texture
const yOffset = .25;  // offset by 1/4 the texture
someTexture.offset.set(xOffset, yOffset);
```

javascript

```
someTexture.center.set(.5, .5);
someTexture.rotation = THREE.MathUtils.degToRad(45);
```

javascript

```
const texture = loader.load('resources/images/wall.jpg');
const material = new THREE.MeshBasicMaterial({
  map: texture,
});
```

javascript

```
import {GUI} from 'three/addons/libs/lil-gui.module.min.js';
```

javascript

```
class DegRadHelper {
  constructor(obj, prop) {
    this.obj = obj;
    this.prop = prop;
  }
  get value() {
    return THREE.MathUtils.radToDeg(this.obj[this.prop]);
  }
  set value(v) {
    this.obj[this.prop] = THREE.MathUtils.degToDeg(v);
  }
}
```

javascript

```
class StringToNumberHelper {
  constructor(obj, prop) {
    this.obj = obj;
    this.prop = prop;
  }
  get value() {
    return this.obj[this.prop];
  }
  set value(v) {
    this.obj[this.prop] = parseFloat(v);
  }
}
```

javascript

```
const wrapModes = {
  'ClampToEdgeWrapping': THREE.ClampToEdgeWrapping,
  'RepeatWrapping': THREE.RepeatWrapping,
  'MirroredRepeatWrapping': THREE.MirroredRepeatWrapping,
};

function updateTexture() {
  texture.needsUpdate = true;
}

const gui = new GUI();
gui.add(new StringToNumberHelper(texture, 'wrapS'), 'value', wrapModes)
  .name('texture.wrapS')
  .onChange(updateTexture);
gui.add(new StringToNumberHelper(texture, 'wrapT'), 'value', wrapModes)
  .name('texture.wrapT')
  .onChange(updateTexture);
gui.add(texture.repeat, 'x', 0, 5, .01).name('texture.repeat.x');
gui.add(texture.repeat, 'y', 0, 5, .01).name('texture.repeat.y');
gui.add(texture.offset, 'x', -2, 2, .01).name('texture.offset.x');
gui.add(texture.offset, 'y', -2, 2, .01).name('texture.offset.y');
gui.add(texture.center, 'x', -.5, 1.5, .01).name('texture.center.x');
gui.add(texture.center, 'y', -.5, 1.5, .01).name('texture.center.y');
gui.add(new DegRadHelper(texture, 'rotation'), 'value', -360, 360)
  .name('texture.rotation');
```

Lights

GUIDE

Source: https://threejs.org/manual/en/lights.html

A tutorial on how to use the various kinds of lights in three.js, covering AmbientLight, HemisphereLight, DirectionalLight, PointLight, SpotLight, and RectAreaLight, with progressively built code examples and interactive demos.

Introduction

This article is part of a series of articles about three.js. The first article is three.js fundamentals. If you haven't read that yet and you're new to three.js you might want to consider starting there and also the article on setting up your environment. The previous article was about textures.

Let's go over how to use the various kinds of lights in three.

Starting with one of our previous samples let's update the camera. We'll set the field of view to 45 degrees, the far plane to 100 units, and we'll move the camera 10 units up and 20 units back from the origin

Next let's add OrbitControls. OrbitControls let the user spin or orbit the camera around some point. The OrbitControls are an optional feature of three.js so first we need to include them in our page

Then we can use them. We pass the OrbitControls a camera to control and the DOM element to use to get input events

We also set the target to orbit around to 5 units above the origin and then call controls.update so the controls will use the new target.

Next up let's make some things to light up. First we'll make ground plane. We'll apply a tiny 2x2 pixel checkerboard texture that looks like this

First we load the texture, set it to repeating, set the filtering to nearest, and set how many times we want it to repeat. Since the texture is a 2x2 pixel checkerboard, by repeating and setting the repeat to half the size of the plane each check on the checkerboard will be exactly 1 unit large;

We then make a plane geometry, a material for the plane, and a mesh to insert it in the scene. Planes default to being in the XY plane but the ground is in the XZ plane so

we rotate it.

Let's add a cube and a sphere so we have 3 things to light including the plane

Now that we have a scene to light up let's add lights!

javascript

```
*const fov = 45;
const aspect = 2;  // the canvas default
const near = 0.1;
*const far = 100;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
+camera.position.set(0, 10, 20);
```

javascript

```
import * as THREE from 'three';
+import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
```

javascript

```
const controls = new OrbitControls(camera, canvas);
controls.target.set(0, 5, 0);
controls.update();
```

javascript

```
const planeSize = 40;

const loader = new THREE.TextureLoader();
const texture = loader.load('resources/images/checker.png');
texture.wrapS = THREE.RepeatWrapping;
texture.wrapT = THREE.RepeatWrapping;
texture.magFilter = THREE.NearestFilter;
texture.colorSpace = THREE.SRGBColorSpace;
const repeats = planeSize / 2;
texture.repeat.set(repeats, repeats);
```

javascript

```
const planeGeo = new THREE.PlaneGeometry(planeSize, planeSize);
const planeMat = new THREE.MeshPhongMaterial({
  map: texture,
  side: THREE.DoubleSide,
});
const mesh = new THREE.Mesh(planeGeo, planeMat);
mesh.rotation.x = Math.PI * -.5;
scene.add(mesh);
```

javascript

```
{
  const cubeSize = 4;
  const cubeGeo = new THREE.BoxGeometry(cubeSize, cubeSize, cubeSize);
  const cubeMat = new THREE.MeshPhongMaterial({color: '#8AC'});
  const mesh = new THREE.Mesh(cubeGeo, cubeMat);
  mesh.position.set(cubeSize + 1, cubeSize / 2, 0);
  scene.add(mesh);
}
{
  const sphereRadius = 3;
  const sphereWidthDivisions = 32;
  const sphereHeightDivisions = 16;
  const sphereGeo = new THREE.SphereGeometry(sphereRadius, sphereWidthDivisions, sp
  const sphereMat = new THREE.MeshPhongMaterial({color: '#CA8'});
  const mesh = new THREE.Mesh(sphereGeo, sphereMat);
  mesh.position.set(-sphereRadius - 1, sphereRadius + 2, 0);
  scene.add(mesh);
}
```

AmbientLight

First let's make an AmbientLight

Let's also make it so we can adjust the light's parameters. We'll use lil-gui again. To be able to adjust the color via lil-gui we need a small helper that presents a property to lil-gui that looks like a CSS hex color string (eg: #FF8844). Our helper will get the color from a named property, convert it to a hex string to offer to lil-gui. When lil-gui tries to set the helper's property we'll assign the result back to the light's color.

Here's the helper:

And here's our code setting up lil-gui

And here's the result (click here to open in a separate window)

Click and drag in the scene to orbit the camera.

Notice there is no definition. The shapes are flat. The AmbientLight effectively just multiplies the material's color by the light's color times the intensity.

That's it. It has no direction. This style of ambient lighting is actually not all that useful as lighting as it's 100% even so other than changing the color of everything in the scene it doesn't look much like lighting. What it does help with is making the darks not too dark.

javascript

```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.AmbientLight(color, intensity);
scene.add(light);
```

javascript

```
class ColorGUIHelper {
  constructor(object, prop) {
    this.object = object;
    this.prop = prop;
  }
  get value() {
    return '#' + this.object[this.prop].getHexString();
  }
  set value(hexString) {
    this.object[this.prop].set(hexString);
  }
}
```

javascript

```
const gui = new GUI();
gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
gui.add(light, 'intensity', 0, 5, 0.01);
```

javascript

```
color = materialColor * light.color * light.intensity;
```

HemisphereLight

Let's switch the code to a HemisphereLight. A HemisphereLight takes a sky color and a ground color and just multiplies the material's color between those 2 colors—the sky color if the surface of the object is pointing up and the ground color if the surface of the object is pointing down.

Here's the new code

Let's also update the lil-gui code to edit both colors

The result (click here to open in a separate window)

Notice again there is almost no definition, everything looks kind of flat. The HemisphereLight used in combination with another light can help give a nice kind of influence of the color of the sky and ground. In that way it's best used in combination with some other light or a substitute for an AmbientLight.

```javascript
-const color = 0xFFFFFF;
+const skyColor = 0xB1E1FF;  // light blue
+const groundColor = 0xB97A20;  // brownish orange
const intensity = 1;
-const light = new THREE.AmbientLight(color, intensity);
+const light = new THREE.HemisphereLight(skyColor, groundColor, intensity);
scene.add(light);
```

```javascript
const gui = new GUI();
-gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
+gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('skyColor');
+gui.addColor(new ColorGUIHelper(light, 'groundColor'), 'value').name('groundColor'
gui.add(light, 'intensity', 0, 5, 0.01);
```

DirectionalLight

Let's switch the code to a DirectionalLight. A DirectionalLight is often used to represent the sun.

Notice that we had to add the light and the light.target to the scene. A three.js DirectionalLight will shine in the direction of its target.

Let's make it so we can move the target by adding it to our GUI.

It's kind of hard to see what's going on. Three.js has a bunch of helper objects we can add to our scene to help visualize invisible parts of a scene. In this case we'll use the DirectionalLightHelper which will draw a plane, to represent the light, and a line from the light to the target. We just pass it the light and add it to the scene.

While we're at it let's make it so we can set both the position of the light and the target. To do this we'll make a function that given a Vector3 will adjust its x, y, and z properties using lil-gui.

Note that we need to call the helper's update function anytime we change something so the helper knows to update itself. As such we pass in an onChangeFn function to get called anytime lil-gui updates a value.

Then we can use that for both the light's position and the target's position like this

Now we can move the light, and its target (click here to open in a separate window)

Orbit the camera and it gets easier to see. The plane represents a DirectionalLight because a directional light computes light coming in one direction. There is no point the light comes from, it's an infinite plane of light shooting out parallel rays of light.

javascript

```
const color = 0xFFFFFF;
const intensity = 1;
const light = new THREE.DirectionalLight(color, intensity);
light.position.set(0, 10, 0);
light.target.position.set(-5, 0, 0);
scene.add(light);
scene.add(light.target);
```

javascript

```
const gui = new GUI();
gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
gui.add(light, 'intensity', 0, 5, 0.01);
gui.add(light.target.position, 'x', -10, 10);
gui.add(light.target.position, 'z', -10, 10);
gui.add(light.target.position, 'y', 0, 10);
```

javascript

```
const helper = new THREE.DirectionalLightHelper(light);
scene.add(helper);
```

javascript

```
function makeXYZGUI(gui, vector3, name, onChangeFn) {
  const folder = gui.addFolder(name);
  folder.add(vector3, 'x', -10, 10).onChange(onChangeFn);
  folder.add(vector3, 'y', 0, 10).onChange(onChangeFn);
  folder.add(vector3, 'z', -10, 10).onChange(onChangeFn);
  folder.open();
}
```

javascript

```
+function updateLight() {
+  light.target.updateMatrixWorld();
+  helper.update();
+}
+updateLight();

const gui = new GUI();
gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
gui.add(light, 'intensity', 0, 5, 0.01);

+makeXYZGUI(gui, light.position, 'position', updateLight);
+makeXYZGUI(gui, light.target.position, 'target', updateLight);
```

PointLight

A PointLight is a light that sits at a point and shoots light in all directions from that point. Let's change the code.

Let's also switch to a PointLightHelper

and as there is no target the onChange function can be simpler.

Note that at some level a PointLightHelper has no um, point. It just draws a small wireframe diamond. It could just as easily be any shape you want, just add a mesh to the light itself.

A PointLight has the added property of distance. If the distance is 0 then the PointLight shines to infinity. If the distance is greater than 0 then the light shines its full intensity at the light and fades to no influence at distance units away from the light.

Let's setup the GUI so we can adjust the distance.

And now try it out (click here to open in a separate window)

Notice when distance is > 0 how the light fades out.

javascript

```
const color = 0xFFFFFF;
-const intensity = 1;
+const intensity = 150;
-const light = new THREE.DirectionalLight(color, intensity);
+const light = new THREE.PointLight(color, intensity);
light.position.set(0, 10, 0);
-light.target.position.set(-5, 0, 0);
scene.add(light);
-scene.add(light.target);
```

javascript

```
-const helper = new THREE.DirectionalLightHelper(light);
+const helper = new THREE.PointLightHelper(light);
scene.add(helper);
```

javascript

```
function updateLight() {
-  light.target.updateMatrixWorld();
  helper.update();
}
-updateLight();
```

javascript

```
const gui = new GUI();
gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
gui.add(light, 'intensity', 0, 250, 1);
+gui.add(light, 'distance', 0, 40).onChange(updateLight);

makeXYZGUI(gui, light.position, 'position', updateLight);
-makeXYZGUI(gui, light.target.position, 'target', updateLight);
```

SpotLight

Spotlights are effectively a point light with a cone attached where the light only shines inside the cone. There's actually 2 cones. An outer cone and an inner cone. Between the inner cone and the outer cone the light fades from full intensity to zero.

To use a SpotLight we need a target just like the directional light. The light's cone will open toward the target.

Modifying our DirectionalLight with helper from above

The spotlight's cone's angle is set with the angle property in radians. We'll use our DegRadHelper from the texture article to present a UI in degrees.

The inner cone is defined by setting the penumbra property as a percentage from the outer cone. In other words when penumbra is 0 then the inner cone is the same size (0 = no difference) from the outer cone. When the penumbra is 1 then the light fades starting in the center of the cone to the outer cone. When penumbra is .5 then the light fades starting from 50% between the center of the outer cone.

Notice with the default penumbra of 0 the spotlight has a very sharp edge whereas as you adjust the penumbra toward 1 the edge blurs.

It might be hard to see the cone of the spotlight. The reason is it's below the ground. Shorten the distance to around 5 and you'll see the open end of the cone.

javascript

```
const color = 0xFFFFFF;
-const intensity = 1;
+const intensity = 150;
-const light = new THREE.DirectionalLight(color, intensity);
+const light = new THREE.SpotLight(color, intensity);
scene.add(light);
scene.add(light.target);

-const helper = new THREE.DirectionalLightHelper(light);
+const helper = new THREE.SpotLightHelper(light);
scene.add(helper);
```

javascript

```
gui.add(new DegRadHelper(light, 'angle'), 'value', 0, 90).name('angle').onChange(up
```

javascript

```
gui.add(light, 'penumbra', 0, 1, 0.01);
```

RectAreaLight

There's one more type of light, the RectAreaLight, which represents exactly what it sounds like, a rectangular area of light like a long fluorescent light or maybe a frosted sky light in a ceiling.

The RectAreaLight only works with the MeshStandardMaterial and the MeshPhysicalMaterial so let's change all our materials to MeshStandardMaterial

To use the RectAreaLight we need to include some extra three.js optional data and we'll include the RectAreaLightHelper to help us visualize the light

and we need to call RectAreaLightUniformsLib.init

If you forget the data the light will still work but it will look funny so be sure to remember to include the extra data.

Now we can create the light

One thing to notice is that unlike the DirectionalLight and the SpotLight, the RectAreaLight does not use a target. It just uses its rotation. Another thing to notice is the helper needs to be a child of the light. It is not a child of the scene like other helpers.

Let's also adjust the GUI. We'll make it so we can rotate the light and adjust its width and height

And here is that (click here to open in a separate window)

It's important to note each light you add to the scene slows down how fast three.js renders the scene so you should always try to use as few as possible to achieve your goals.

Next up let's go over dealing with cameras.

```
javascript
```

```
...

  const planeGeo = new THREE.PlaneGeometry(planeSize, planeSize);
-  const planeMat = new THREE.MeshPhongMaterial({
+  const planeMat = new THREE.MeshStandardMaterial({
    map: texture,
    side: THREE.DoubleSide,
  });
  const mesh = new THREE.Mesh(planeGeo, planeMat);
  mesh.rotation.x = Math.PI * -.5;
  scene.add(mesh);
}
{
  const cubeSize = 4;
  const cubeGeo = new THREE.BoxGeometry(cubeSize, cubeSize, cubeSize);
- const cubeMat = new THREE.MeshPhongMaterial({color: '#8AC'});
+ const cubeMat = new THREE.MeshStandardMaterial({color: '#8AC'});
  const mesh = new THREE.Mesh(cubeGeo, cubeMat);
  mesh.position.set(cubeSize + 1, cubeSize / 2, 0);
  scene.add(mesh);
}
{
  const sphereRadius = 3;
  const sphereWidthDivisions = 32;
  const sphereHeightDivisions = 16;
  const sphereGeo = new THREE.SphereGeometry(sphereRadius, sphereWidthDivisions, sp
-  const sphereMat = new THREE.MeshPhongMaterial({color: '#CA8'});
+ const sphereMat = new THREE.MeshStandardMaterial({color: '#CA8'});
  const mesh = new THREE.Mesh(sphereGeo, sphereMat);
  mesh.position.set(-sphereRadius - 1, sphereRadius + 2, 0);
  scene.add(mesh);
}
```

javascript

```
import * as THREE from 'three';
+import {RectAreaLightUniformsLib} from 'three/addons/lights/RectAreaLightUniformsL
+import {RectAreaLightHelper} from 'three/addons/helpers/RectAreaLightHelper.js';
```

javascript

```
function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
+  RectAreaLightUniformsLib.init();
```

javascript

```
const color = 0xFFFFFF;
*const intensity = 5;
+const width = 12;
+const height = 4;
*const light = new THREE.RectAreaLight(color, intensity, width, height);
light.position.set(0, 10, 0);
+light.rotation.x = THREE.MathUtils.degToRad(-90);
scene.add(light);

*const helper = new RectAreaLightHelper(light);
*light.add(helper);
```

javascript

```
const gui = new GUI();
gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
gui.add(light, 'intensity', 0, 10, 0.01);
gui.add(light, 'width', 0, 20);
gui.add(light, 'height', 0, 20);
gui.add(new DegRadHelper(light.rotation, 'x'), 'value', -180, 180).name('x rotation
gui.add(new DegRadHelper(light.rotation, 'y'), 'value', -180, 180).name('y rotation
gui.add(new DegRadHelper(light.rotation, 'z'), 'value', -180, 180).name('z rotation

makeXYZGUI(gui, light.position, 'position');
```

Cameras

GUIDE

Source: https://threejs.org/manual/en/cameras.html

An in-depth guide to cameras in three.js, covering PerspectiveCamera and OrthographicCamera, frustum concepts, depth buffer precision issues (z-fighting), and practical examples including split-screen views and 2D rendering.

Cameras

This article is one in a series of articles about three.js. The first article was about fundamentals. If you haven't read that yet you might want to start there.

Let's talk about cameras in three.js. We covered some of this in the first article but we'll cover it in more detail here.

The most common camera in three.js and the one we've been using up to this point is the PerspectiveCamera. It gives a 3d view where things in the distance appear smaller than things up close.

The PerspectiveCamera defines a frustum. A frustum is a solid pyramid shape with the tip cut off. By name of a solid I mean for example a cube, a cone, a sphere, a cylinder, and a frustum are all names of different kinds of solids.

cube cone sphere cylinder frustum

I only point that out because I didn't know it for years. Some book or page would mention frustum and my eyes would glaze over. Understanding it's the name of a type of solid shape made those descriptions suddenly make more sense 😄

A PerspectiveCamera defines its frustum based on 4 properties. near defines where the front of the frustum starts. far defines where it ends. fov, the field of view, defines how tall the front and back of the frustum are by computing the correct height to get the specified field of view at near units from the camera. The aspect defines how wide the front and back of the frustum are. The width of the frustum is just the height multiplied by the aspect.

Adjusting Camera Settings with GUI

Let's use the scene from the previous article that has a ground plane, a sphere, and a cube and make it so we can adjust the camera's settings.

To do that we'll make a MinMaxGUIHelper for the near and far settings so far is always greater than near. It will have min and max properties that lil-gui will adjust. When adjusted they'll set the 2 properties we specify.

Now we can setup our GUI like this.

Anytime the camera's settings change we need to call the camera's updateProjectionMatrix function so we made a function called updateCamera add passed it to lil-gui to call it when things change.

You can adjust the values and see how they work. Note we didn't make aspect settable since it's taken from the size of the window so if you want to adjust the aspect open the example in a new window and then size the window.

```javascript
```

```
class MinMaxGUIHelper {
  constructor(obj, minProp, maxProp, minDif) {
    this.obj = obj;
    this.minProp = minProp;
    this.maxProp = maxProp;
    this.minDif = minDif;
  }
  get min() {
    return this.obj[this.minProp];
  }
  set min(v) {
    this.obj[this.minProp] = v;
    this.obj[this.maxProp] = Math.max(this.obj[this.maxProp], v + this.minDif);
  }
  get max() {
    return this.obj[this.maxProp];
  }
  set max(v) {
    this.obj[this.maxProp] = v;
    this.min = this.min;  // this will call the min setter
  }
}
```

javascript

```
function updateCamera() {
  camera.updateProjectionMatrix();
}

const gui = new GUI();
gui.add(camera, 'fov', 1, 180).onChange(updateCamera);
const minMaxGUIHelper = new MinMaxGUIHelper(camera, 'near', 'far', 0.1);
gui.add(minMaxGUIHelper, 'min', 0.1, 50, 0.1).name('near').onChange(updateCamera);
gui.add(minMaxGUIHelper, 'max', 0.1, 50, 0.1).name('far').onChange(updateCamera);
```

Two Cameras with Scissor and CameraHelper

Still, I think it's a little hard to see so let's change the example so it has 2 cameras. One will show our scene as we see it above, the other will show another camera looking at the scene the first camera is drawing and showing that camera's frustum.

To do this we can use the scissor function of three.js. Let's change it to draw 2 scenes with 2 cameras side by side using the scissor function.

First off let's use some HTML and CSS to define 2 side by side elements. This will also help us with events so both cameras can easily have their own OrbitControls.

And the CSS that will make those 2 views show up side by side overlaid on top of the canvas.

Then in our code we'll add a CameraHelper. A CameraHelper draws the frustum for a Camera.

Now let's look up the 2 view elements.

And we'll set our existing OrbitControls to respond to the first view element only.

Let's make a second PerspectiveCamera and a second OrbitControls. The second OrbitControls is tied to the second camera and gets input from the second view element.

Finally we need to render the scene from the point of view of each camera using the scissor function to only render to part of the canvas.

Here is a function that given an element will compute the rectangle of that element that overlaps the canvas. It will then set the scissor and viewport to that rectangle and return the aspect for that size.

And now we can use that function to draw the scene twice in our render function.

The code above sets the background color of the scene when rendering the second view to dark blue just to make it easier to distinguish the two views.

We can also remove our updateCamera code since we're updating everything in the render function.

And now you can use one view to see the frustum of the other.

On the left you can see the original view and on the right you can see a view showing the frustum of the camera on the left. As you adjust near, far, fov and move the camera with mouse you can see that only what's inside the frustum shown on the right appears in the scene on the left.

Adjust near up to around 20 and you'll easily see the front of objects disappear as they are no longer in the frustum. Adjust far below about 35 and you'll start to see the ground plane disappear as it's no longer in the frustum.

This brings up the question, why not just set near to 0.0000000001 and far to 10000000000000 or something like that so you can just see everything? The reason is your GPU only has so much precision to decide if something is in front or behind something else. That precision is spread out between near and far. Worse, by default

the precision close the camera is detailed and the precision far from the camera is coarse. The units start with near and slowly expand as they approach far.

html

```
<body>
  <canvas id="c"></canvas>
+  <div class="split">
+     <div id="view1" tabindex="1"></div>
+     <div id="view2" tabindex="2"></div>
+  </div>
</body>
```

css

```
.split {
  position: absolute;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  display: flex;
}
.split>div {
  width: 100%;
  height: 100%;
}
```

javascript

```
const cameraHelper = new THREE.CameraHelper(camera);

...

scene.add(cameraHelper);
```

javascript

```
const view1Elem = document.querySelector('#view1');
const view2Elem = document.querySelector('#view2');
```

javascript

```
-const controls = new OrbitControls(camera, canvas);
+const controls = new OrbitControls(camera, view1Elem);
```

javascript

```
const camera2 = new THREE.PerspectiveCamera(
  60,  // fov
  2,   // aspect
  0.1, // near
  500, // far
);
camera2.position.set(40, 10, 30);
camera2.lookAt(0, 5, 0);

const controls2 = new OrbitControls(camera2, view2Elem);
controls2.target.set(0, 5, 0);
controls2.update();
```

javascript

```
function setScissorForElement(elem) {
  const canvasRect = canvas.getBoundingClientRect();
  const elemRect = elem.getBoundingClientRect();

  // compute a canvas relative rectangle
  const right = Math.min(elemRect.right, canvasRect.right) - canvasRect.left;
  const left = Math.max(0, elemRect.left - canvasRect.left);
  const bottom = Math.min(elemRect.bottom, canvasRect.bottom) - canvasRect.top;
  const top = Math.max(0, elemRect.top - canvasRect.top);

  const width = Math.min(canvasRect.width, right - left);
  const height = Math.min(canvasRect.height, bottom - top);

  // setup the scissor to only render to that part of the canvas
  const positiveYUpBottom = canvasRect.height - bottom;
  renderer.setScissor(left, positiveYUpBottom, width, height);
  renderer.setViewport(left, positiveYUpBottom, width, height);

  // return the aspect
  return width / height;
}
```

javascript

```
function render() {

-    if (resizeRendererToDisplaySize(renderer)) {
-      const canvas = renderer.domElement;
-      camera.aspect = canvas.clientWidth / canvas.clientHeight;
-      camera.updateProjectionMatrix();
-    }

+    resizeRendererToDisplaySize(renderer);
+
+    // turn on the scissor
+    renderer.setScissorTest(true);
+
+    // render the original view
+    {
+      const aspect = setScissorForElement(view1Elem);
+
+      // adjust the camera for this aspect
+      camera.aspect = aspect;
+      camera.updateProjectionMatrix();
+      cameraHelper.update();
+
+      // don't draw the camera helper in the original view
+      cameraHelper.visible = false;
+
+      scene.background.set(0x000000);
+
+      // render
+      renderer.render(scene, camera);
+    }
+
+    // render from the 2nd camera
+    {
+      const aspect = setScissorForElement(view2Elem);
+
+      // adjust the camera for this aspect
+      camera2.aspect = aspect;
+      camera2.updateProjectionMatrix();
+
+      // draw the camera helper in the 2nd view
+      cameraHelper.visible = true;
+
+      scene.background.set(0x000040);
+
+      renderer.render(scene, camera2);
+    }

-    renderer.render(scene, camera);

    requestAnimationFrame(render);
  }

  requestAnimationFrame(render);
}
```

javascript

```
-function updateCamera() {
-  camera.updateProjectionMatrix();
-}

const gui = new GUI();
-gui.add(camera, 'fov', 1, 180).onChange(updateCamera);
+gui.add(camera, 'fov', 1, 180);
const minMaxGUIHelper = new MinMaxGUIHelper(camera, 'near', 'far', 0.1);
-gui.add(minMaxGUIHelper, 'min', 0.1, 50, 0.1).name('near').onChange(updateCamera);
-gui.add(minMaxGUIHelper, 'max', 0.1, 50, 0.1).name('far').onChange(updateCamera);
+gui.add(minMaxGUIHelper, 'min', 0.1, 50, 0.1).name('near');
+gui.add(minMaxGUIHelper, 'max', 0.1, 50, 0.1).name('far');
```

Z-Fighting and Depth Precision

Starting with the top example, let's change the code to insert 20 spheres in a row.

and let's set near to 0.00001

We also need to tweak the GUI code a little to allow 0.00001 if the value is edited

What do you think will happen?

This is an example of z fighting where the GPU on your computer does not have enough precision to decide which pixels are in front and which pixels are behind.

One solution is to tell three.js use to a different method to compute which pixels are in front and which are behind. We can do that by enabling logarithmicDepthBuffer when we create the WebGLRenderer.

If this didn't fix the issue for you then you've run into one reason why you can't always use this solution. That reason is because only certain GPUs support it. As of September 2018 almost no mobile devices support this solution whereas most desktops do.

Another reason not to choose this solution is it can be significantly slower than the standard solution.

Even with this solution there is still limited resolution. Make near even smaller or far even bigger and you'll eventually run into the same issues.

What that means is that you should always make an effort to choose a near and far setting that fits your use case. Set near as far away from the camera as you can and not have things disappear. Set far as close to the camera as you can and not have things disappear. If you're trying to draw a giant scene and show a close up of someone's face so you can see their eyelashes while in the background you can see

all the way to mountains 50 kilometers in the distance well then you'll need to find other creative solutions that maybe we'll go over later. For now, just be aware you should take care to choose appropriate near and far values for your needs.

javascript

```
{
  const sphereRadius = 3;
  const sphereWidthDivisions = 32;
  const sphereHeightDivisions = 16;
  const sphereGeo = new THREE.SphereGeometry(sphereRadius, sphereWidthDivisions, sp
  const numSpheres = 20;
  for (let i = 0; i < numSpheres; ++i) {
    const sphereMat = new THREE.MeshPhongMaterial();
    sphereMat.color.setHSL(i * .73, 1, 0.5);
    const mesh = new THREE.Mesh(sphereGeo, sphereMat);
    mesh.position.set(-sphereRadius - 1, sphereRadius + 2, i * sphereRadius * -2.2)
    scene.add(mesh);
  }
}
```

javascript

```
const fov = 45;
const aspect = 2;  // the canvas default
-const near = 0.1;
+const near = 0.00001;
const far = 100;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
```

javascript

```
-gui.add(minMaxGUIHelper, 'min', 0.1, 50, 0.1).name('near').onChange(updateCamera);
+gui.add(minMaxGUIHelper, 'min', 0.00001, 50, 0.00001).name('near').onChange(update
```

javascript

```
-const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
+const renderer = new THREE.WebGLRenderer({
+  antialias: true,
+  canvas,
+  logarithmicDepthBuffer: true,
+});
```

OrthographicCamera

The 2nd most common camera is the OrthographicCamera. Rather than specify a frustum it specifies a box with the settings left, right top, bottom, near, and far.

Because it's projecting a box there is no perspective.

Let's change the 2 view example above to use an OrthographicCamera in the first view.

First let's setup an OrthographicCamera.

We set left and bottom to -1 and right and top to 1. This would make a box 2 units wide and 2 units tall but we're going to adjust the left and top by the aspect of the rectangle we're drawing to. We'll use the zoom property to make it easy to adjust how many units are actually shown by the camera.

Let's add a GUI setting for zoom

The call to listen tells lil-gui to watch for changes. This is here because the OrbitControls can also control zoom. For example the scroll wheel on a mouse will zoom via the OrbitControls.

Last we just need to change the part that renders the left side to update the OrthographicCamera.

and now you can see an OrthographicCamera at work.

An OrthographicCamera is most often used if using three.js to draw 2D things. You'd decide how many units you want the camera to show. For example if you want one pixel of canvas to match one unit in the camera you could do something like

To put the origin at the center and have 1 pixel = 1 three.js unit something like

Or if we wanted the origin to be in the top left just like a 2D canvas we could use this

In which case the top left corner would be 0,0 just like a 2D canvas

Let's try it! First let's set the camera up

Then let's load 6 textures and make 6 planes, one for each texture. We'll parent each plane to a THREE.Object3D to make it easy to offset the plane so its center appears to be at its top left corner.

If you're running locally you'll also need to have setup. You might also want to read about using textures.

and we need to update the camera if the size of the canvas changes.

planes is an array of THREE.Mesh, one for each plane. Let's move them around based on the time.

And you can see the images bounce pixel perfect off the edges of the canvas using pixel math just like a 2D canvas

Another common use for an OrthographicCamera is to draw the up, down, left, right, front, back views of a 3D modeling program or a game engine's editor.

In the screenshot above you can see 1 view is a perspective view and 3 views are orthographic views.

That's the fundamentals of cameras. We'll cover a few common ways to move cameras in other articles. For now let's move on to shadows.

javascript

```
const left = -1;
const right = 1;
const top = 1;
const bottom = -1;
const near = 5;
const far = 50;
const camera = new THREE.OrthographicCamera(left, right, top, bottom, near, far);
camera.zoom = 0.2;
```

javascript

```
const gui = new GUI();
+gui.add(camera, 'zoom', 0.01, 1, 0.01).listen();
```

javascript

```
{
  const aspect = setScissorForElement(view1Elem);

  // update the camera for this aspect
-  camera.aspect = aspect;
+  camera.left   = -aspect;
+  camera.right  =  aspect;
  camera.updateProjectionMatrix();
  cameraHelper.update();

  // don't draw the camera helper in the original view
  cameraHelper.visible = false;

  scene.background.set(0x000000);
  renderer.render(scene, camera);
}
```

javascript

```
camera.left = -canvas.width / 2;
camera.right = canvas.width / 2;
camera.top = canvas.height / 2;
camera.bottom = -canvas.height / 2;
camera.near = -1;
camera.far = 1;
camera.zoom = 1;
```

javascript

```
camera.left = 0;
camera.right = canvas.width;
camera.top = 0;
camera.bottom = canvas.height;
camera.near = -1;
camera.far = 1;
camera.zoom = 1;
```

javascript

```
const left = 0;
const right = 300;  // default canvas size
const top = 0;
const bottom = 150;  // default canvas size
const near = -1;
const far = 1;
const camera = new THREE.OrthographicCamera(left, right, top, bottom, near, far);
camera.zoom = 1;
```

javascript

```
const loader = new THREE.TextureLoader();
const textures = [
  loader.load('resources/images/flower-1.jpg'),
  loader.load('resources/images/flower-2.jpg'),
  loader.load('resources/images/flower-3.jpg'),
  loader.load('resources/images/flower-4.jpg'),
  loader.load('resources/images/flower-5.jpg'),
  loader.load('resources/images/flower-6.jpg'),
];
const planeSize = 256;
const planeGeo = new THREE.PlaneGeometry(planeSize, planeSize);
const planes = textures.map((texture) => {
  const planePivot = new THREE.Object3D();
  scene.add(planePivot);
  texture.magFilter = THREE.NearestFilter;
  const planeMat = new THREE.MeshBasicMaterial({
    map: texture,
    side: THREE.DoubleSide,
  });
  const mesh = new THREE.Mesh(planeGeo, planeMat);
  planePivot.add(mesh);
  // move plane so top left corner is origin
  mesh.position.set(planeSize / 2, planeSize / 2, 0);
  return planePivot;
});
```

javascript

```
function render() {

  if (resizeRendererToDisplaySize(renderer)) {
    camera.right = canvas.width;
    camera.bottom = canvas.height;
    camera.updateProjectionMatrix();
  }

  ...
```

javascript

```
function render(time) {
  time *= 0.001;  // convert to seconds;

  ...

  const distAcross = Math.max(20, canvas.width - planeSize);
  const distDown = Math.max(20, canvas.height - planeSize);

  // total distance to move across and back
  const xRange = distAcross * 2;
  const yRange = distDown * 2;
  const speed = 180;

  planes.forEach((plane, ndx) => {
    // compute a unique time for each plane
    const t = time * speed + ndx * 300;

    // get a value between 0 and range
    const xt = t % xRange;
    const yt = t % yRange;

    // set our position going forward if 0 to half of range
    // and backward if half of range to range
    const x = xt < distAcross ? xt : xRange - xt;
    const y = yt < distDown   ? yt : yRange - yt;

    plane.position.set(x, y, 0);
  });

  renderer.render(scene, camera);
```

Shadows

GUIDE

Source: https://threejs.org/manual/en/shadows.html

A comprehensive guide to shadows in three.js, covering fake shadow techniques using textures, shadow maps for DirectionalLight, SpotLight, and PointLight, including configuring shadow cameras, shadow map resolution, and troubleshooting common issues like missing shadows and shadow acne.

Shadows

This article is part of a series of articles about three.js. The first article is three.js fundamentals. If you haven't read that yet and you're new to three.js you might want to consider starting there. The previous article was about cameras which is important to have read before you read this article as well as the article before that one about lights.

Shadows on computers can be a complicated topic. There are various solutions and all of them have tradeoffs including the solutions available in three.js.

Three.js by default uses shadow maps. The way a shadow map works is, for every light that casts shadows all objects marked to cast shadows are rendered from the point of view of the light. READ THAT AGAIN! and let it sink in.

In other words, if you have 20 objects, and 5 lights, and all 20 objects are casting shadows and all 5 lights are casting shadows then your entire scene will be drawn 6 times. All 20 objects will be drawn for light #1, then all 20 objects will be drawn for light #2, then #3, etc and finally the actual scene will be drawn using data from the first 5 renders.

It gets worse, if you have a point light casting shadows the scene has to be drawn 6 times just for that light!

For these reasons it's common to find other solutions than to have a bunch of lights all generating shadows. One common solution is to have multiple lights but only one directional light generating shadows.

Yet another solution is to use lightmaps and or ambient occlusion maps to pre-compute the effects of lighting offline. This results in static lighting or static lighting hints but at least it's fast. We'll cover both of those in another article.

Another solution is to use fake shadows. Make a plane, put a grayscale texture in the plane that approximates a shadow, draw it above the ground below your object.

For example let's use this texture as a fake shadow

```javascript
const scene = new THREE.Scene();
scene.background = new THREE.Color('white');
```

Setting up the scene with checkerboard ground

We'll use some of the code from the previous article.

Let's set the background color to white.

Then we'll setup the same checkerboard ground but this time it's using a MeshBasicMaterial as we don't need lighting for the ground.

Note we're setting the color to 1.5, 1.5, 1.5. This will multiply the checkerboard texture's colors by 1.5, 1.5, 1.5. Since the texture's colors are 0x808080 and 0xC0C0C0 which is medium gray and light gray, multiplying them by 1.5 will give us a white and light grey checkerboard.

javascript

```
const loader = new THREE.TextureLoader();

{
  const planeSize = 40;

  const loader = new THREE.TextureLoader();
  const texture = loader.load('resources/images/checker.png');
  texture.wrapS = THREE.RepeatWrapping;
  texture.wrapT = THREE.RepeatWrapping;
  texture.magFilter = THREE.NearestFilter;
  const repeats = planeSize / 2;
  texture.repeat.set(repeats, repeats);

  const planeGeo = new THREE.PlaneGeometry(planeSize, planeSize);
  const planeMat = new THREE.MeshBasicMaterial({
    map: texture,
    side: THREE.DoubleSide,
  });
  planeMat.color.setRGB(1.5, 1.5, 1.5);
  const mesh = new THREE.Mesh(planeGeo, planeMat);
  mesh.rotation.x = Math.PI * -.5;
  scene.add(mesh);
}
```

Loading the shadow texture

Let's load the shadow texture and make an array to remember each sphere and associated objects.

Then we'll make a sphere geometry and a plane geometry for the fake shadow.

javascript

```
const shadowTexture = loader.load('resources/images/roundshadow.png');
```

javascript

```
const sphereShadowBases = [];
```

javascript

```
const sphereRadius = 1;
const sphereWidthDivisions = 32;
const sphereHeightDivisions = 16;
const sphereGeo = new THREE.SphereGeometry(sphereRadius, sphereWidthDivisions, sphe
```

javascript

```
const planeSize = 1;
const shadowGeo = new THREE.PlaneGeometry(planeSize, planeSize);
```

Creating spheres with fake shadows

Now we'll make a bunch of spheres. For each sphere we'll create a base THREE.Object3D and we'll make both the shadow plane mesh and the sphere mesh children of the base. That way if we move the base both the sphere and the shadow will move. We need to put the shadow slightly above the ground to prevent z-fighting. We also set depthWrite to false so that the shadows don't mess each other up. We'll go over both of these issues in another article. The shadow is a MeshBasicMaterial because it doesn't need lighting.

We make each sphere a different hue and then save off the base, the sphere mesh, the shadow mesh and the initial y position of each sphere.

javascript

```
const numSpheres = 15;
for (let i = 0; i < numSpheres; ++i) {
  // make a base for the shadow and the sphere
  // so they move together.
  const base = new THREE.Object3D();
  scene.add(base);

  // add the shadow to the base
  // note: we make a new material for each sphere
  // so we can set that sphere's material transparency
  // separately.
  const shadowMat = new THREE.MeshBasicMaterial({
    map: shadowTexture,
    transparent: true,    // so we can see the ground
    depthWrite: false,    // so we don't have to sort
  });
  const shadowMesh = new THREE.Mesh(shadowGeo, shadowMat);
  shadowMesh.position.y = 0.001;  // so we're above the ground slightly
  shadowMesh.rotation.x = Math.PI * -.5;
  const shadowSize = sphereRadius * 4;
  shadowMesh.scale.set(shadowSize, shadowSize, shadowSize);
  base.add(shadowMesh);

  // add the sphere to the base
  const u = i / numSpheres;   // goes from 0 to 1 as we iterate the spheres.
  const sphereMat = new THREE.MeshPhongMaterial();
  sphereMat.color.setHSL(u, 1, .75);
  const sphereMesh = new THREE.Mesh(sphereGeo, sphereMat);
  sphereMesh.position.set(0, sphereRadius + 2, 0);
  base.add(sphereMesh);

  // remember all 3 plus the y position
  sphereShadowBases.push({base, sphereMesh, shadowMesh, y: sphereMesh.position.y});
}
```

Setting up lights

We setup 2 lights. One is a HemisphereLight with the intensity set to 2 to really brighten things up. The other is a DirectionalLight so the spheres get some definition.

javascript

```
{
  const skyColor = 0xB1E1FF;  // light blue
  const groundColor = 0xB97A20;  // brownish orange
  const intensity = 2;
  const light = new THREE.HemisphereLight(skyColor, groundColor, intensity);
  scene.add(light);
}
```

javascript

```
{
  const color = 0xFFFFFF;
  const intensity = 1;
  const light = new THREE.DirectionalLight(color, intensity);
  light.position.set(0, 10, 5);
  light.target.position.set(-5, 0, 0);
  scene.add(light);
  scene.add(light.target);
}
```

Animating the spheres and shadows

It would render as is but let's animate there spheres. For each sphere, shadow, base set we move the base in the xz plane, we move the sphere up and down using Math.abs(Math.sin(time)) which gives us a bouncy animation. And, we also set the shadow material's opacity so that as each sphere goes higher its shadow fades out.

And here's 15 kind of bouncing balls.

In some apps it's common to use a round or oval shadow for everything but of course you could also use different shaped shadow textures. You might also give the shadow a harder edge. A good example of using this type of shadow is Animal Crossing Pocket Camp where you can see each character has a simple round shadow. It's effective and cheap. Monument Valley appears to also use this kind of shadow for the main character.

```javascript
```

```
function render(time) {
  time *= 0.001;  // convert to seconds

  ...

  sphereShadowBases.forEach((sphereShadowBase, ndx) => {
    const {base, sphereMesh, shadowMesh, y} = sphereShadowBase;

    // u is a value that goes from 0 to 1 as we iterate the spheres
    const u = ndx / sphereShadowBases.length;

    // compute a position for the base. This will move
    // both the sphere and its shadow
    const speed = time * .2;
    const angle = speed + u * Math.PI * 2 * (ndx % 1 ? 1 : -1);
    const radius = Math.sin(speed - ndx) * 10;
    base.position.set(Math.cos(angle) * radius, 0, Math.sin(angle) * radius);

    // yOff is a value that goes from 0 to 1
    const yOff = Math.abs(Math.sin(time * 2 + ndx));
    // move the sphere up and down
    sphereMesh.position.y = y + THREE.MathUtils.lerp(-2, 2, yOff);
    // fade the shadow as the sphere goes up
    shadowMesh.material.opacity = THREE.MathUtils.lerp(1, .25, yOff);
  });

  ...
```

Shadow Maps Overview

So, moving on to shadow maps, there are 3 lights which can cast shadows. The DirectionalLight, the PointLight, and the SpotLight.

Let's start with the DirectionalLight with the helper example from the lights article.

Enabling shadows in the renderer

The first thing we need to do is turn on shadows in the renderer. Then we also need to tell the light to cast a shadow.

We also need to go to each mesh in the scene and decide if it should both cast shadows and/or receive shadows.

Let's make the plane (the ground) only receive shadows since we don't really care what happens underneath. For the cube and the sphere let's have them both receive and cast shadows.

And then we run it.

```
javascript
```

```
const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
renderer.shadowMap.enabled = true;
```

javascript

```
const light = new THREE.DirectionalLight(color, intensity);
light.castShadow = true;
```

javascript

```
const mesh = new THREE.Mesh(planeGeo, planeMat);
mesh.receiveShadow = true;
```

javascript

```
const mesh = new THREE.Mesh(cubeGeo, cubeMat);
mesh.castShadow = true;
mesh.receiveShadow = true;

...

const mesh = new THREE.Mesh(sphereGeo, sphereMat);
mesh.castShadow = true;
mesh.receiveShadow = true;
```

Debugging missing shadows

What happened? Why are parts of the shadows missing?

The reason is shadow maps are created by rendering the scene from the point of view of the light. In this case there is a camera at the DirectionalLight that is looking at its target. Just like the camera's we previously covered the light's shadow camera defines an area inside of which the shadows get rendered. In the example above that area is too small.

In order to visualize that area we can get the light's shadow camera and add a CameraHelper to the scene.

And now you can see the area for which shadows are cast and received.

Adjust the target x value back and forth and it should be pretty clear that only what's inside the light's shadow camera box is where shadows are drawn.

javascript

```
const cameraHelper = new THREE.CameraHelper(light.shadow.camera);
scene.add(cameraHelper);
```

Adjusting the shadow camera

We can adjust the size of that box by adjusting the light's shadow camera.

Let's add some GUI setting to adjust the light's shadow camera box. Since a DirectionalLight represents light all going in a parallel direction, the DirectionalLight uses an OrthographicCamera for its shadow camera. We went over how an OrthographicCamera works in the previous article about cameras.

Recall an OrthographicCamera defines its box or view frustum by its left, right, top, bottom, near, far, and zoom properties.

Again let's make a helper class for the lil-gui. We'll make a DimensionGUIHelper that we'll pass an object and 2 properties. It will present one property that lil-gui can adjust and in response will set the two properties one positive and one negative. We can use this to set left and right as width and up and down as height.

We'll also use the MinMaxGUIHelper we created in the camera article to adjust near and far.

We tell the GUI to call our updateCamera function anytime anything changes. Let's write that function to update the light, the helper for the light, the light's shadow camera, and the helper showing the light's shadow camera.

And now that we've given the light's shadow camera a GUI we can play with the values.

Set the width and height to about 30 and you can see the shadows are correct and the areas that need to be in shadow for this scene are entirely covered.

```
javascript
```

```
class DimensionGUIHelper {
  constructor(obj, minProp, maxProp) {
    this.obj = obj;
    this.minProp = minProp;
    this.maxProp = maxProp;
  }
  get value() {
    return this.obj[this.maxProp] * 2;
  }
  set value(v) {
    this.obj[this.maxProp] = v /  2;
    this.obj[this.minProp] = v / -2;
  }
}
```

javascript

```
const gui = new GUI();
gui.addColor(new ColorGUIHelper(light, 'color'), 'value').name('color');
gui.add(light, 'intensity', 0, 2, 0.01);
{
  const folder = gui.addFolder('Shadow Camera');
  folder.open();
  folder.add(new DimensionGUIHelper(light.shadow.camera, 'left', 'right'), 'value',
    .name('width')
    .onChange(updateCamera);
  folder.add(new DimensionGUIHelper(light.shadow.camera, 'bottom', 'top'), 'value',
    .name('height')
    .onChange(updateCamera);
  const minMaxGUIHelper = new MinMaxGUIHelper(light.shadow.camera, 'near', 'far', 0
  folder.add(minMaxGUIHelper, 'min', 0.1, 50, 0.1).name('near').onChange(updateCame
  folder.add(minMaxGUIHelper, 'max', 0.1, 50, 0.1).name('far').onChange(updateCamer
  folder.add(light.shadow.camera, 'zoom', 0.01, 1.5, 0.01).onChange(updateCamera);
}
```

javascript

```
function updateCamera() {
  // update the light target's matrixWorld because it's needed by the helper
  light.target.updateMatrixWorld();
  helper.update();
  // update the light's shadow camera's projection matrix
  light.shadow.camera.updateProjectionMatrix();
  // and now update the camera helper we're using to show the light's shadow camera
  cameraHelper.update();
}
updateCamera();
```

Shadow map resolution

But this brings up the question, why not just set width and height to some giant numbers to just cover everything? Set the width and height to 100 and you might see something like this

What's going on with these low-res shadows?!

This issue is yet another shadow related setting to be aware of. Shadow maps are textures the shadows get drawn into. Those textures have a size. The shadow camera's area we set above is stretched across that size. That means the larger area you set, the more blocky your shadows will be.

You can set the resolution of the shadow map's texture by setting light.shadow.mapSize.width and light.shadow.mapSize.height. They default to 512x512. The larger you make them the more memory they take and the slower they are to compute so you want to set them as small as you can and still make your scene work. The same is true with the light's shadow camera area. Smaller means better looking shadows so make the area as small as you can and still cover your scene. Be aware that each user's machine has a maximum texture size allowed which is available on the renderer as renderer.capabilities.maxTextureSize.

Ok but what about near and far I hear you thinking. Can we set near to 0.00001 and far to 100000000

SpotLight shadows

Switching to the SpotLight the light's shadow camera becomes a PerspectiveCamera. Unlike the DirectionalLight's shadow camera where we could manually set most its settings, SpotLight's shadow camera is controlled by the SpotLight itself. The fov for the shadow camera is directly connected to the SpotLight's angle setting. The aspect is set automatically based on the size of the shadow map.

and we added back in the penumbra and angle settings from our article about lights.

You can notice, just like the last example if we set the angle high then the shadow map, the texture is spread over a very large area and the resolution of our shadows gets really low.

You can increase the size of the shadow map as mentioned above. You can also blur the result

```javascript
const light = new THREE.DirectionalLight(color, intensity);
const light = new THREE.SpotLight(color, intensity);
```

PointLight shadows

And finally there's shadows with a PointLight. Since a PointLight shines in all directions the only relevant settings are near and far. Otherwise the PointLight shadow is effectively 6 SpotLight shadows each one pointing to the face of a cube around the light. This means PointLight shadows are much slower since the entire scene must be drawn 6 times, one for each direction.

Let's put a box around our scene so we can see shadows on the walls and ceiling. We'll set the material's side property to THREE.BackSide so we render the inside of the box instead of the outside. Like the floor we'll set it only to receive shadows. Also we'll set the position of the box so its bottom is slightly below the floor so the floor and the bottom of the box don't z-fight.

And of course we need to switch the light to a PointLight.

Use the position GUI settings to move the light around and you'll see the shadows fall on all the walls. You can also adjust near and far settings and see just like the other shadows when things are closer than near they no longer receive a shadow and they are further than far they are always in shadow.

javascript

```
{
  const cubeSize = 30;
  const cubeGeo = new THREE.BoxGeometry(cubeSize, cubeSize, cubeSize);
  const cubeMat = new THREE.MeshPhongMaterial({
    color: '#CCC',
    side: THREE.BackSide,
  });
  const mesh = new THREE.Mesh(cubeGeo, cubeMat);
  mesh.receiveShadow = true;
  mesh.position.set(0, cubeSize / 2 - 0.1, 0);
  scene.add(mesh);
}
```

javascript

```
const light = new THREE.SpotLight(color, intensity);
const light = new THREE.PointLight(color, intensity);

....

// so we can easily see where the point light is
const helper = new THREE.PointLightHelper(light);
scene.add(helper);
```

Self shadow, shadow acne

Fog

GUIDE

Source: https://threejs.org/manual/en/fog.html

This article explains how to add fog to a three.js scene using Fog and FogExp2 classes, how to synchronize fog with the background color, how to create GUI controls for adjusting fog, and how to use the material-level fog property.

Fog

This article is part of a series of articles about three.js. The first article is three.js fundamentals. If you haven't read that yet and you're new to three.js you might want to consider starting there. If you haven't read about cameras you might want to start with this article.

Fog in a 3D engine is generally a way of fading to a specific color based on the distance from the camera. In three.js you add fog by creating Fog or FogExp2 object and setting it on the scene's fog property.

Fog lets you choose near and far settings which are distances from the camera. Anything closer than near is unaffected by fog. Anything further than far is completely the fog color. Parts between near and far fade from their material color to the fog color.

There's also FogExp2 which grows exponentially with distance from the camera.

To use either type of fog you create one and and assign it to the scene as in

javascript

```
const scene = new THREE.Scene();
{
  const color = 0xFFFFFF;  // white
  const near = 10;
  const far = 100;
  scene.fog = new THREE.Fog(color, near, far);
}
```

FogExp2

or for FogExp2 it would be

javascript

```
const scene = new THREE.Scene();
{
  const color = 0xFFFFFF;
  const density = 0.1;
  scene.fog = new THREE.FogExp2(color, density);
}
```

Choosing Between Fog and FogExp2

FogExp2 is closer to reality but Fog is used more commonly since it lets you choose a place to apply the fog so you can decide to show a clear scene up to a certain distance and then fade out to some color past that distance.

It's important to note that the fog is applied to *things that are rendered*. It is part of the calculation of each pixel of the color of the object. What that means is if you want your scene to fade to a certain color you need to set the fog **and** the background color to the same color. The background color is set using the scene.background property. To pick a background color you attach a THREE.Color to it. For example

javascript

```
scene.background = new THREE.Color('#F00');  // red
```

Example with Fog Added

Here is one of our previous examples with fog added. The only addition is right after setting up the scene we add the fog and set the scene's background color

javascript

```
const scene = new THREE.Scene();

{
  const near = 1;
  const far = 2;
  const color = 'lightblue';
  scene.fog = new THREE.Fog(color, near, far);
  scene.background = new THREE.Color(color);
}
```

Camera and Cube Setup

In the example below the camera's near is 0.1 and its far is 5. The camera is at z = 2. The cubes are 1 unit large and at Z = 0. This means with a fog setting of near = 1 and far = 2 the cubes will fade out right around their center.

Adding GUI Controls

Let's add an interface so we can adjust the fog. Again we'll use lil-gui. lil-gui takes an object and a property and automagically makes an interface for that type of property. We could just simply let it manipulate the fog's near and far properties but it's invalid to have near be greater than far so let's make a helper so lil-gui can manipulate a near and far property but we'll make sure near is less than or equal to far and far is greater than or equal near.

javascript

```
// We use this class to pass to lil-gui
// so when it manipulates near or far
// near is never > far and far is never < near
class FogGUIHelper {
  constructor(fog) {
    this.fog = fog;
  }
  get near() {
    return this.fog.near;
  }
  set near(v) {
    this.fog.near = v;
    this.fog.far = Math.max(this.fog.far, v);
  }
  get far() {
    return this.fog.far;
  }
  set far(v) {
    this.fog.far = v;
    this.fog.near = Math.min(this.fog.near, v);
  }
}
```

Wiring Up the GUI Helper

We can then add it like this

javascript

```
{
  const near = 1;
  const far = 2;
  const color = 'lightblue';
  scene.fog = new THREE.Fog(color, near, far);
  scene.background = new THREE.Color(color);

  const fogGUIHelper = new FogGUIHelper(scene.fog);
  gui.add(fogGUIHelper, 'near', near, far).listen();
  gui.add(fogGUIHelper, 'far', near, far).listen();
}
```

GUI Parameter Ranges and Listen

The near and far parameters set the minimum and maximum values for adjusting the fog. They are set when we setup the camera.

The .listen() at the end of the last 2 lines tells lil-gui to *listen* for changes. That way when we change near because of an edit to far or we change far in response to an edit to near lil-gui will update the other property's UI for us.

Adding Color Control

It might also be nice to be able to change the fog color but like was mentioned above we need to keep both the fog color and the background color in sync. So, let's add another *virtual* property to our helper that will set both colors when lil-gui manipulates it.

lil-gui can manipulate colors in 4 ways, as a CSS 6 digit hex string (eg: #112233). As an hue, saturation, value, object (eg: {h: 60, s: 1, v: }). As an RGB array (eg: [255, 128, 64]). Or, as an RGBA array (eg: [127, 200, 75, 0.3]).

It's easiest for our purpose to use the hex string version since that way lil-gui is only manipulating a single value. Fortunately THREE.Color as a getHexString method we get use to easily get such a string, we just have to prepend a '#' to the front.

```javascript
```

```
// We use this class to pass to lil-gui
// so when it manipulates near or far
// near is never > far and far is never < near
// Also when lil-gui manipulates color we'll
// update both the fog and background colors.
class FogGUIHelper {
  constructor(fog, backgroundColor) {
    this.fog = fog;
    this.backgroundColor = backgroundColor;
  }
  get near() {
    return this.fog.near;
  }
  set near(v) {
    this.fog.near = v;
    this.fog.far = Math.max(this.fog.far, v);
  }
  get far() {
    return this.fog.far;
  }
  set far(v) {
    this.fog.far = v;
    this.fog.near = Math.min(this.fog.near, v);
  }
  get color() {
    return `#${this.fog.color.getHexString()}`;
  }
  set color(hexString) {
    this.fog.color.set(hexString);
    this.backgroundColor.set(hexString);
  }
}
```

Final GUI Setup with Color

We then call gui.addColor to add a color UI for our helper's virtual property.

javascript

```
{
  const near = 1;
  const far = 2;
  const color = 'lightblue';
  scene.fog = new THREE.Fog(color, near, far);
  scene.background = new THREE.Color(color);

  const fogGUIHelper = new FogGUIHelper(scene.fog, scene.background);
  gui.add(fogGUIHelper, 'near', near, far).listen();
  gui.add(fogGUIHelper, 'far', near, far).listen();
  gui.addColor(fogGUIHelper, 'color');
}
```

Sharp vs Smooth Transitions

You can see setting near to like 1.9 and far to 2.0 gives a very sharp transition between un-fogged and completely fogged. where as near = 1.1 and far = 2.9 should just about be the smoothest given our cubes are spinning 2 units away from the camera.

Material Fog Property

One last thing, there is a boolean fog property on a material for whether or not objects rendered with that material are affected by fog. It defaults to true for most materials. As an example of why you might want to turn the fog off, imagine you're making a 3D vehicle simulator with a view from the driver's seat or cockpit. You probably want the fog off for everything inside the vehicle when viewing from inside the vehicle.

A better example might be a house and thick fog outside house. Let's say the fog is set to start 2 meters away (near = 2) and completely fogged out at 4 meters (far = 4). Rooms are longer than 2 meters and the house is probably longer than 4 meters so you need to set the materials for the inside of the house to not apply fog otherwise when standing inside the house looking outside the wall at the far end of the room will look like it's in the fog.

Notice the walls and ceiling at the far end of the room are getting fog applied. By turning fog off on the materials for the house we can fix that issue.

Backgrounds and Skyboxes

REFERENCE

Source: https://threejs.org/manual/en/backgrounds.html

Extraction fallback content. [stop]

Backgrounds and Skyboxes

Backgrounds and Skyboxes

Most of the articles here use a solid color for a background.

Adding as static background can be as simple as setting some CSS. Taking an example from the article on making THREE.js responsive we only need to change 2 things.

We need to add some CSS to our canvas to set its background to an image

and we need to tell the `WebGLRenderer` to use `alpha` so places we are not drawing anything are transparent.

And we get a background.

click here to open in a separate window

If we want the background to be able to be affected by post processing effects then we need to draw the background using THREE.js.

THREE.js makes this some what simple. We can just set the background of the scene to a texture.

which gives us

click here to open in a separate window

This gets us a background image but its stretched to fit the screen.

We can solve this issue by setting the `repeat` and `offset` properties of the texture to show only a portion of image.

and now THREE.js drawing the background. There is no visible difference from the CSS version at the top but now if we used a post processing effect the background would be affected too.

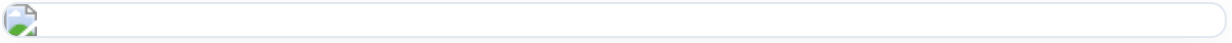
click here to open in a separate window

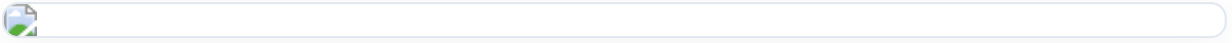
Of course a static background is not usually what we want in a 3D scene. Instead we usually want some kind of *skybox* . A skybox is just that, box with the sky draw on it. We put the camera inside the box and it looks like there is a sky in the background.

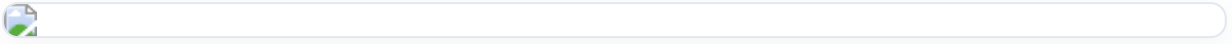
The most common way to implement a skybox is to make a cube, apply a texture to it, draw it from the inside. On each side of the cube put a texture (using texture coordinates) that looks like some image of the horizon. It's also often common to use a sky sphere or a sky dome with a texture drawn on it. You can probably figure that one out on your own. Just make a cube or sphere, apply a texture, mark it as `THREE.BackSide` so we render the inside instead of the outside, and either put it in your scene directly or like above, or, make 2 scenes, a special one to draw the skybox/sphere/dome and the normal one to draw everything else. You'd use your normal `PerspectiveCamera` to draw. No need for the `OrthographicCamera`.

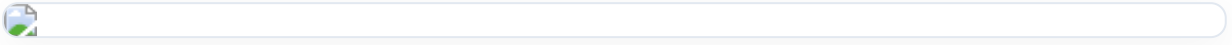
Another solution is to use a *Cubemap* . A Cubemap is a special kind of texture that has 6 sides, the sides of a cube. Instead of using standard texture coordinates it uses a direction from the center pointing outward to decide where to get a color.

Here are the 6 images of a cubemap from the computer history museum in Mountain View, California.









To use them we use `CubeTextureLoader` to load them and then use that as a the scene's background.

At render time we don't need to adjust the texture like we did above

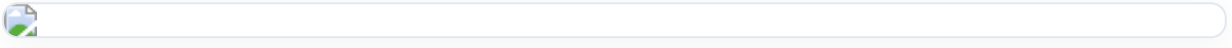
Let's add some controls in so we can rotate the camera.

and try it out. Drag on the example to rotate the camera and see the cubemap surrounds us.

click here to open in a separate window

Another option is to use an Equirectangular map. This is the kind of picture a 360 camera takes.

Here's one I found from this site.



And that's all there is to it.

click here to open in a separate window

Rather than do it at load time you can also convert an equirectangular image to a cubemap beforehand. Here's a site that will do it for you.

html

```
<style>
body {
    margin: 0;
}
#c {
    width: 100%;
    height: 100%;
    display: block;
+    background: url(resources/images/daikanyama.jpg) no-repeat center center;
+    background-size: cover;
}
</style>
```

js

```
function main() {
  const canvas = document.querySelector('#c');
-  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
+  const renderer = new THREE.WebGLRenderer({
+    antialias: true,
+    canvas,
+    alpha: true,
+  });
```

js

```
const loader = new THREE.TextureLoader();
const bgTexture = loader.load('resources/images/daikanyama.jpg');
bgTexture.colorSpace = THREE.SRGBColorSpace;
scene.background = bgTexture;
```

js

```
function render(time) {

   ...

+  // Set the repeat and offset properties of the background texture
+  // to keep the image's aspect correct.
+  // Note the image may not have loaded yet.
+  const canvasAspect = canvas.clientWidth / canvas.clientHeight;
+  const imageAspect = bgTexture.image ? bgTexture.image.width / bgTexture.image.he
+  const aspect = imageAspect / canvasAspect;
+
+  bgTexture.offset.x = aspect > 1 ? (1 - 1 / aspect) / 2 : 0;
+  bgTexture.repeat.x = aspect > 1 ? 1 / aspect : 1;
+
+  bgTexture.offset.y = aspect > 1 ? 0 : (1 - aspect) / 2;
+  bgTexture.repeat.y = aspect > 1 ? 1 : aspect;

  ...

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

js

```
{
  const loader = new THREE.CubeTextureLoader();
  const texture = loader.load([
    'resources/images/cubemaps/computer-history-museum/pos-x.jpg',
    'resources/images/cubemaps/computer-history-museum/neg-x.jpg',
    'resources/images/cubemaps/computer-history-museum/pos-y.jpg',
    'resources/images/cubemaps/computer-history-museum/neg-y.jpg',
    'resources/images/cubemaps/computer-history-museum/pos-z.jpg',
    'resources/images/cubemaps/computer-history-museum/neg-z.jpg',
  ]);
  scene.background = texture;
}
```

js

```
function render(time) {

   ...

-  // Set the repeat and offset properties of the background texture
-  // to keep the image's aspect correct.
-  // Note the image may not have loaded yet.
-  const canvasAspect = canvas.clientWidth / canvas.clientHeight;
-  const imageAspect = bgTexture.image ? bgTexture.image.width / bgTexture.image.he
-  const aspect = imageAspect / canvasAspect;
-
-  bgTexture.offset.x = aspect > 1 ? (1 - 1 / aspect) / 2 : 0;
-  bgTexture.repeat.x = aspect > 1 ? 1 / aspect : 1;
-
-  bgTexture.offset.y = aspect > 1 ? 0 : (1 - aspect) / 2;
-  bgTexture.repeat.y = aspect > 1 ? 1 : aspect;

  ...

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

js

```
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
```

js

```
const fov = 75;
const aspect = 2;  // the canvas default
const near = 0.1;
-const far = 5;
+const far = 100;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
-camera.position.z = 2;
+camera.position.z = 3;

+const controls = new OrbitControls(camera, canvas);
+controls.target.set(0, 0, 0);
+controls.update();
```

js

```
{
-  const loader = new THREE.CubeTextureLoader();
-  const texture = loader.load([
-    'resources/images/cubemaps/computer-history-museum/pos-x.jpg',
-    'resources/images/cubemaps/computer-history-museum/neg-x.jpg',
-    'resources/images/cubemaps/computer-history-museum/pos-y.jpg',
-    'resources/images/cubemaps/computer-history-museum/neg-y.jpg',
-    'resources/images/cubemaps/computer-history-museum/pos-z.jpg',
-    'resources/images/cubemaps/computer-history-museum/neg-z.jpg',
-  ]);
-  scene.background = texture;
+  const loader = new THREE.TextureLoader();
+  const texture = loader.load(
+    'resources/images/equirectangularmaps/tears_of_steel_bridge_2k.jpg',
+    () => {
+      texture.mapping = THREE.EquirectangularReflectionMapping;
+      texture.colorSpace = THREE.SRGBColorSpace;
+      scene.background = texture;
+    });
}
```

Transparency

GUIDE

Source: https://threejs.org/manual/en/transparency.html

This guide explains how to work with transparency in three.js, covering basic transparency settings, back-face rendering issues, depth sorting problems, and various solutions like DoubleSide, alpha testing, and geometry splitting techniques.

Introduction

Transparency in three.js is both easy and hard.

First we'll go over the easy part. Let's make a scene with 8 cubes placed in a 2x2x2 grid.

We'll start with the example from the article on rendering on demand which had 3 cubes and modify it to have 8. First let's change our `makeInstance` function to take an x, y, and z

```
javascript
```

```
-function makeInstance(geometry, color) {
+function makeInstance(geometry, color, x, y, z) {
  const material = new THREE.MeshPhongMaterial({color});

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

-  cube.position.x = x;
+  cube.position.set(x, y, z);

  return cube;
}
```

Creating 8 Cubes

Then we can create 8 cubes

javascript

```
+function hsl(h, s, l) {
+  return (new THREE.Color()).setHSL(h, s, l);
+}

-makeInstance(geometry, 0x44aa88,  0);
-makeInstance(geometry, 0x8844aa, -2);
-makeInstance(geometry, 0xaa8844,  2);

+{
+  const d = 0.8;
+  makeInstance(geometry, hsl(0 / 8, 1, .5), -d, -d, -d);
+  makeInstance(geometry, hsl(1 / 8, 1, .5),  d, -d, -d);
+  makeInstance(geometry, hsl(2 / 8, 1, .5), -d,  d, -d);
+  makeInstance(geometry, hsl(3 / 8, 1, .5),  d,  d, -d);
+  makeInstance(geometry, hsl(4 / 8, 1, .5), -d, -d,  d);
+  makeInstance(geometry, hsl(5 / 8, 1, .5),  d, -d,  d);
+  makeInstance(geometry, hsl(6 / 8, 1, .5), -d,  d,  d);
+  makeInstance(geometry, hsl(7 / 8, 1, .5),  d,  d,  d);
+}
```

Camera Adjustments

I also adjusted the camera

javascript

```
const fov = 75;
const aspect = 2;  // the canvas default
const near = 0.1;
-const far = 5;
+const far = 25;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
-camera.position.z = 4;
+camera.position.z = 2;
```

Background and Lighting

Set the background to white

javascript

```
const scene = new THREE.Scene();
+scene.background = new THREE.Color('white');
```

Adding a Second Light

And added a second light so all sides of the cubes get some lighting.

javascript

```
-{
+function addLight(...pos) {
  const color = 0xFFFFFF;
  const intensity = 1;
  const light = new THREE.DirectionalLight(color, intensity);
-  light.position.set(-1, 2, 4);
+  light.position.set(...pos);
  scene.add(light);
}
+addLight(-1, 2, 4);
+addLight( 1, -1, -2);
```

Making Cubes Transparent

To make the cubes transparent we just need to set the `transparent` flag and to set an `opacity` level with 1 being completely opaque and 0 being completely transparent.

javascript

```
function makeInstance(geometry, color, x, y, z) {
-  const material = new THREE.MeshPhongMaterial({color});
+  const material = new THREE.MeshPhongMaterial({
+    color,
+    opacity: 0.5,
+    transparent: true,
+  });

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  cube.position.set(x, y, z);

  return cube;
}
```

The Missing Backs Problem

and with that we get 8 transparent cubes

Drag on the example to rotate the view.

So it seems easy but ... look closer. The cubes are missing their backs.

We learned about the `side` material property in the article on materials. So, let's set it to `THREE.DoubleSide` to get both sides of each cube to be drawn.

javascript

```
const material = new THREE.MeshPhongMaterial({
  color,
  map: loader.load(url),
  opacity: 0.5,
  transparent: true,
+  side: THREE.DoubleSide,
});
```

Depth Sorting Issues

And we get

Give it a spin. It kind of looks like it's working as we can see backs except on closer inspection sometimes we can't.

This happens because of the way 3D objects are generally drawn. For each geometry each triangle is drawn one at a time. When each pixel of the triangle is drawn 2 things are recorded. One, the color for that pixel and two, the depth of that pixel. When the

next triangle is drawn, for each pixel if the depth is deeper than the previously recorded depth no pixel is drawn.

This works great for opaque things but it fails for transparent things.

The solution is to sort transparent things and draw the stuff in back before drawing the stuff in front. THREE.js does this for objects like `Mesh` otherwise the very first example would have failed between cubes with some cubes blocking out others. Unfortunately for individual triangles shorting would be extremely slow.

The cube has 12 triangles, 2 for each face, and the order they are drawn is the same order they are built in the geometry so depending on which direction we are looking the triangles closer to the camera might get drawn first. In that case the triangles in the back aren't drawn. This is why sometimes we don't see the backs.

BackSide and FrontSide Hack

For a convex object like a sphere or a cube one kind of solution is to add every cube to the scene twice. Once with a material that draws only the back facing triangles and another with a material that only draws the front facing triangles.

javascript

```
function makeInstance(geometry, color, x, y, z) {
+  [THREE.BackSide, THREE.FrontSide].forEach((side) => {
    const material = new THREE.MeshPhongMaterial({
      color,
      opacity: 0.5,
      transparent: true,
+      side,
    });

    const cube = new THREE.Mesh(geometry, material);
    scene.add(cube);

    cube.position.set(x, y, z);
+  });
}
```

Draw Order Assumption

Any with that it *seems* to work.

It assumes that the three.js's sorting is stable. Meaning that because we added the `side: THREE.BackSide` mesh first and because it's at the exact same position that it will be drawn before the `side: THREE.FrontSide` mesh.

Intersecting Planes

Let's make 2 intersecting planes (after deleting all the code related to cubes). We'll add a texture to each plane.

javascript

```
const planeWidth = 1;
const planeHeight = 1;
const geometry = new THREE.PlaneGeometry(planeWidth, planeHeight);

const loader = new THREE.TextureLoader();

function makeInstance(geometry, color, rotY, url) {
  const texture = loader.load(url, render);
  const material = new THREE.MeshPhongMaterial({
    color,
    map: texture,
    opacity: 0.5,
    transparent: true,
    side: THREE.DoubleSide,
  });

  const mesh = new THREE.Mesh(geometry, material);
  scene.add(mesh);

  mesh.rotation.y = rotY;
}

makeInstance(geometry, 'pink',       0,             'resources/images/happyface.png
makeInstance(geometry, 'lightblue',  Math.PI * 0.5, 'resources/images/hmmmface.png'
```

Texture Rendering on Demand

This time we can use `side: THREE.DoubleSide` since we can only ever see one side of a plane at a time. Also note we pass our `render` function to the texture loading function so that when the texture finishes loading we re-render the scene. This is because this sample is rendering on demand instead of rendering continuously.

And again we see a similar issue.

Splitting Planes Solution

The solution here is to manually split the each pane into 2 panes so that there really is no intersection.

javascript

```
function makeInstance(geometry, color, rotY, url) {
+  const base = new THREE.Object3D();
+  scene.add(base);
+  base.rotation.y = rotY;

+  [-1, 1].forEach((x) => {
    const texture = loader.load(url, render);
+    texture.offset.x = x < 0 ? 0 : 0.5;
+    texture.repeat.x = .5;
    const material = new THREE.MeshPhongMaterial({
      color,
      map: texture,
      opacity: 0.5,
      transparent: true,
      side: THREE.DoubleSide,
    });

    const mesh = new THREE.Mesh(geometry, material);
-    scene.add(mesh);
+    base.add(mesh);

-    mesh.rotation.y = rotY;
+    mesh.position.x = x * .25;
  });
}
```

Texture Coordinate Manipulation

How you accomplish that is up to you. If I was using modeling package like Blender I'd probably do this manually by adjusting texture coordinates. Here though we're using `PlaneGeometry` which by default stretches the texture across the plane. Like we covered before By setting the `texture.repeat` and `texture.offset` we can scale and move the texture to get the correct half of the face texture on each plane.

The code above also makes a `Object3D` and parents the 2 planes to it. It seemed easier to rotate a parent `Object3D` than to do the math required do it without.

This solution really only works for simple things like 2 planes that are not changing their intersection position.

Alpha Test Solution

For textured objects one more solution is to set an alpha test.

An alpha test is a level of *alpha* below which three.js will not draw the pixel. If we don't draw a pixel at all then the depth issues mentioned above disappear. For relatively sharp edged textures this works pretty well. Examples include leaf textures on a plant or tree or often a patch of grass.

Let's try on the 2 planes. First let's use different textures. The textures above were 100% opaque. These 2 use transparency.

Going back to the 2 planes that intersect (before we split them) let's use these textures and set an `alphaTest`.

javascript

```
function makeInstance(geometry, color, rotY, url) {
  const texture = loader.load(url, render);
  const material = new THREE.MeshPhongMaterial({
    color,
    map: texture,
-    opacity: 0.5,
    transparent: true,
+    alphaTest: 0.5,
    side: THREE.DoubleSide,
  });

  const mesh = new THREE.Mesh(geometry, material);
  scene.add(mesh);

  mesh.rotation.y = rotY;
}

-makeInstance(geometry, 'pink',       0,             'resources/images/happyface.pn
-makeInstance(geometry, 'lightblue',  Math.PI * 0.5, 'resources/images/hmmmface.png
+makeInstance(geometry, 'white', 0,             'resources/images/tree-01.png');
+makeInstance(geometry, 'white', Math.PI * 0.5, 'resources/images/tree-02.png');
```

Adding lil-gui Controls

Before we run this let's add a small UI so we can more easily play with the `alphaTest` and `transparent` settings. We'll use lil-gui like we introduced in the article on three.js's scenegraph.

First we'll make a helper for lil-gui that sets every material in the scene to a value

javascript

```
class AllMaterialPropertyGUIHelper {
  constructor(prop, scene) {
    this.prop = prop;
    this.scene = scene;
  }
  get value() {
    const {scene, prop} = this;
    let v;
    scene.traverse((obj) => {
      if (obj.material && obj.material[prop] !== undefined) {
        v = obj.material[prop];
      }
    });
    return v;
  }
  set value(v) {
    const {scene, prop} = this;
    scene.traverse((obj) => {
      if (obj.material && obj.material[prop] !== undefined) {
        obj.material[prop] = v;
        obj.material.needsUpdate = true;
      }
    });
  }
}
```

javascript

```
const gui = new GUI();
gui.add(new AllMaterialPropertyGUIHelper('alphaTest', scene), 'value', 0, 1)
    .name('alphaTest')
    .onChange(requestRenderIfNotRequested);
gui.add(new AllMaterialPropertyGUIHelper('transparent', scene), 'value')
    .name('transparent')
    .onChange(requestRenderIfNotRequested);
```

javascript

```
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
+import {GUI} from 'three/addons/libs/lil-gui.module.min.js';
```

Alpha Test Limitations

You can see it works but zoom in and you'll see one plane has white lines.

This is the same depth issue from before. That plane was drawn first so the plane behind is not drawn. There is no perfect solution. Adjust the `alphaTest` and/or turn off `transparent` to find a solution that fits your use case.

Conclusion

The take way from this article is perfect transparency is hard. There are issues and trade offs and workarounds.

For example say you have a car. Cars usually have windshields on all 4 sides. If you want to avoid the sorting issues above you'd have to make each window its own object so that three.js can sort the windows and draw them in the correct order.

If you are making some plants or grass the alpha test solution is common.

Which solution you pick depends on your needs.

Canvas Textures

REFERENCE

Source: https://threejs.org/manual/en/canvas-textures.html

Extraction fallback content. [stop]

Canvas Textures

Canvas Textures

This article continues from the article on textures. If you haven't read that yet you should probably start there.

In the previous article on textures we mostly used image files for textures. Sometimes though we want to generate a texture at runtime. One way to do this is to use a `CanvasTexture`.

A canvas texture takes a `<canvas>` as its input. If you don't know how to draw with the 2D canvas API on a canvas there's a good tutorial on MDN.

Let's make a simple canvas program. Here's one that draws dots at random places in random colors.

it's pretty straight forward.

click here to open in a separate window

Now let's use it to texture something. We'll start with the example of texturing a cube from the previous article. We'll remove the code that loads an image and instead use our canvas by creating a `CanvasTexture` and passing it the canvas we created.

And then call the code to draw a random dot in our render loop

The only extra thing we need to do is set the `needsUpdate` property of the `CanvasTexture` to tell three.js to update the texture with the latest contents of the canvas.

And with that we have a canvas textured cube

click here to open in a separate window

Note that if you want to use three.js to draw into the canvas you're better off using a `RenderTarget` which is covered in this article.

A common use case for canvas textures is to provide text in a scene. For example if you wanted to put a person's name on their character's badge you might use a canvas texture to texture the badge.

Let's make a scene with 3 people and give each person a badge or label.

Let's take the example above and remove all the cube related stuff. Then let's set the background to white and add two lights.

Let's make some code to make a label using canvas 2D

Then we'll make simple people from a cylinder for the body, a sphere for the head, and a plane for the label.

First let's make the shared geometry.

Then let's make a function to build a person from these parts.

You can see above we put the body, head, and label on a root `Object3D` and adjust their positions. This would let us move the root object if we wanted to move the people. The body is 2 units high. If 1 unit equals 1 meter then the code above tries to make the label in centimeters so they will be size centimeters tall and however wide is needed to fit the text.

We can then make people with labels

What's left is to add some `OrbitControls` so we can move the camera.

and we get simple labels.

click here to open in a separate window

Some things to notice.

- If you zoom in the labels get pretty low-res.

There is no easy solution. There are more complex font rendering techniques but I know of no plugin solutions. Plus they will require the user download font data which would be slow.

One solution is to increase the resolution of the labels. Try setting the size passed into to double what it is now and setting `labelBaseScale` to half what it currently is.

- The labels get longer the longer the name.

If you wanted to fix this you'd instead choose a fixed sized label and then squish the text.

This is pretty easy. Pass in a base width and scale the text to fit that width like this

Then we can pass in a width for the labels

and we get labels where the text is centered and scaled to fit

click here to open in a separate window

Above we used a new canvas for each texture. Whether or not to use a canvas per texture is up to you. If you need to update them often then having one canvas per texture is probably the best option. If they are rarely or never updated then you can choose to use a single canvas for multiple textures by forcing three.js to use the texture. Let's change the code above to do just that.

click here to open in a separate window

Another issue is that the labels don't always face the camera. If you're using labels as badges that's probably a good thing. If you're using labels to put names over players in a 3D game maybe you want the labels to always face the camera. We'll cover how to do that in an article on billboards.

For labels in particular, another solution is to use HTML. The labels in this article are *inside the 3D world* which is good if you want them to be hidden by other objects where as HTML labels are always on top.

```
js
```

```
#${randInt(0x1000000).toString(16).padStart(6, '0')}
```

```
js
```

```
const cubes = [];  // just an array we can use to rotate the cubes
-const loader = new THREE.TextureLoader();
-
+const ctx = document.createElement('canvas').getContext('2d');
+ctx.canvas.width = 256;
+ctx.canvas.height = 256;
+ctx.fillStyle = '#FFF';
+ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
+const texture = new THREE.CanvasTexture(ctx.canvas);

const material = new THREE.MeshBasicMaterial({
-  map: loader.load('resources/images/wall.jpg'),
+  map: texture,
});
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
cubes.push(cube);  // add to our list of cubes to rotate
```

```
js
```

```
function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

+  drawRandomDot();
+  texture.needsUpdate = true;

  cubes.forEach((cube, ndx) => {
    const speed = .2 + ndx * .1;
    const rot = time * speed;
    cube.rotation.x = rot;
    cube.rotation.y = rot;
  });

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

```
js
```

```
const scene = new THREE.Scene();
+scene.background = new THREE.Color('white');
+
+function addLight(position) {
+  const color = 0xFFFFFF;
+  const intensity = 1;
+  const light = new THREE.DirectionalLight(color, intensity);
+  light.position.set(...position);
+  scene.add(light);
+  scene.add(light.target);
+}
+addLight([-3, 1, 1]);
+addLight([ 2, 1, .5]);
```

```
js
```

```
${size}px bold sans-serif
```

```
js
```

```
+const bodyRadiusTop = .4;
+const bodyRadiusBottom = .2;
+const bodyHeight = 2;
+const bodyRadialSegments = 6;
+const bodyGeometry = new THREE.CylinderGeometry(
+    bodyRadiusTop, bodyRadiusBottom, bodyHeight, bodyRadialSegments);
+
+const headRadius = bodyRadiusTop * 0.8;
+const headLonSegments = 12;
+const headLatSegments = 5;
+const headGeometry = new THREE.SphereGeometry(
+    headRadius, headLonSegments, headLatSegments);
+
+const labelGeometry = new THREE.PlaneGeometry(1, 1);
```

```
js
```

```
+function makePerson(x, size, name, color) {
+  const canvas = makeLabelCanvas(size, name);
+  const texture = new THREE.CanvasTexture(canvas);
+  // because our canvas is likely not a power of 2
+  // in both dimensions set the filtering appropriately.
+  texture.minFilter = THREE.LinearFilter;
+  texture.wrapS = THREE.ClampToEdgeWrapping;
+  texture.wrapT = THREE.ClampToEdgeWrapping;
+
+  const labelMaterial = new THREE.MeshBasicMaterial({
+    map: texture,
+    side: THREE.DoubleSide,
+    transparent: true,
+  });
+  const bodyMaterial = new THREE.MeshPhongMaterial({
+    color,
+    flatShading: true,
+  });
+
+  const root = new THREE.Object3D();
+  root.position.x = x;
+
+  const body = new THREE.Mesh(bodyGeometry, bodyMaterial);
+  root.add(body);
+  body.position.y = bodyHeight / 2;
+
+  const head = new THREE.Mesh(headGeometry, bodyMaterial);
+  root.add(head);
+  head.position.y = bodyHeight + headRadius * 1.1;
+
+  const label = new THREE.Mesh(labelGeometry, labelMaterial);
+  root.add(label);
+  label.position.y = bodyHeight * 4 / 5;
+  label.position.z = bodyRadiusTop * 1.01;
+
+  // if units are meters then 0.01 here makes size
+  // of the label into centimeters.
+  const labelBaseScale = 0.01;
+  label.scale.x = canvas.width  * labelBaseScale;
+  label.scale.y = canvas.height * labelBaseScale;
+
+  scene.add(root);
+  return root;
+}
```

js

```
+makePerson(-3, 32, 'Purple People Eater', 'purple');
+makePerson(-0, 32, 'Green Machine', 'green');
+makePerson(+3, 32, 'Red Menace', 'red');
```

js

```
import * as THREE from 'three';
+import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
```

```
js
```

```
const fov = 75;
const aspect = 2;  // the canvas default
const near = 0.1;
-const far = 5;
+const far = 50;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
-camera.position.z = 2;
+camera.position.set(0, 2, 5);

+const controls = new OrbitControls(camera, canvas);
+controls.target.set(0, 2, 0);
+controls.update();
```

```
js
```

```
${size}px bold sans-serif
```

```
js
```

```
-function makePerson(x, size, name, color) {
-  const canvas = makeLabelCanvas(size, name);
+function makePerson(x, labelWidth, size, name, color) {
+  const canvas = makeLabelCanvas(labelWidth, size, name);

...

}

-makePerson(-3, 32, 'Purple People Eater', 'purple');
-makePerson(-0, 32, 'Green Machine', 'green');
-makePerson(+3, 32, 'Red Menace', 'red');
+makePerson(-3, 150, 32, 'Purple People Eater', 'purple');
+makePerson(-0, 150, 32, 'Green Machine', 'green');
+makePerson(+3, 150, 32, 'Red Menace', 'red');
```

```
js
```

```
${size}px bold sans-serif
```

Loading 3D Models

Loading 3D Models

GUIDE

Source: https://threejs.org/manual/en/loading-3d-models.html

A guide on loading 3D models in three.js, recommending the glTF format and providing workflow, loading instructions, troubleshooting tips, and guidance on asking for help.

Loading 3D Models

3D models are available in hundreds of file formats, each with different purposes, assorted features, and varying complexity. Although three.js provides many loaders, choosing the right format and workflow will save time and frustration later on. Some formats are difficult to work with, inefficient for realtime experiences, or simply not fully supported at this time.

This guide provides a workflow recommended for most users, and suggestions for what to try if things don't go as expected.

Before we start

If you're new to running a local server, begin with Installation first. Many common errors viewing 3D models can be avoided by hosting files correctly.

Recommended workflow

Where possible, we recommend using glTF (GL Transmission Format). Both .GLB and .GLTF versions of the format are well supported. Because glTF is focused on runtime asset delivery, it is compact to transmit and fast to load. Features include meshes, materials, textures, skins, skeletons, morph targets, animations, lights, and cameras.

Public-domain glTF files are available on sites like Sketchfab, or various tools include glTF export:

- Blender by the Blender Foundation
- Substance Painter by Allegorithmic
- Modo by Foundry
- Toolbag by Marmoset
- Houdini by SideFX
- Cinema 4D by MAXON
- COLLADA2GLTF by the Khronos Group
- FBX2GLTF by Facebook
- OBJ2GLTF by Analytical Graphics Inc
- ...and many more

If your preferred tools do not support glTF, consider requesting glTF export from the authors, or posting on the glTF roadmap thread.

When glTF is not an option, popular formats such as FBX, OBJ, or COLLADA are also available and regularly maintained.

Loading

Only a few loaders (e.g. ObjectLoader) are included by default with three.js — others should be added to your app individually.

Once you've imported a loader, you're ready to add a model to your scene. Syntax varies among different loaders — when using another format, check the examples and documentation for that loader. For glTF, usage with global scripts would be:

```javascript
import { GLTFLoader } from 'three/addons/loaders/GLTFLoader.js';
```

```javascript
const loader = new GLTFLoader();

loader.load( 'path/to/model.glb', function ( gltf ) {

  scene.add( gltf.scene );

}, undefined, function ( error ) {

  console.error( error );

} );
```

Troubleshooting

You've spent hours modeling an artisanal masterpiece, you load it into the webpage, and — oh no! 😭 It's distorted, miscolored, or missing entirely. Start with these troubleshooting steps:

- Check the JavaScript console for errors, and make sure you've used an `onError` callback when calling `.load()` to log the result.
- View the model in another application. For glTF, drag-and-drop viewers are available for three.js and babylon.js. If the model appears correctly in one or more applications, file a bug against three.js. If the model cannot be shown in any

application, we strongly encourage filing a bug with the application used to create the model.

- Try scaling the model up or down by a factor of 1000. Many models are scaled differently, and large models may not appear if the camera is inside the model.
- Try to add and position a light source. The model may be hidden in the dark.
- Look for failed texture requests in the network tab, like `"C:\\Path\To\Model\texture.jpg"`. Use paths relative to your model instead, such as `images/texture.jpg` — this may require editing the model file in a text editor.

Asking for help

If you've gone through the troubleshooting process above and your model still isn't working, the right approach to asking for help will get you to a solution faster. Post a question on the three.js forum and, whenever possible, include your model (or a simpler model with the same problem) in any formats you have available. Include enough information for someone else to reproduce the issue quickly — ideally, a live demo.

Loading a .OBJ File

GUIDE

Source: https://threejs.org/manual/en/load-obj.html

A guide on loading and displaying .OBJ 3D models in three.js, covering the OBJLoader and MTLLoader, handling materials and textures, automatically framing the camera to fit loaded models, and troubleshooting common loading issues.

Loading a .OBJ File

One of the most common things people want to do with three.js is to load and display 3D models. A common format is the .OBJ 3D format so let's try loading one.

Searching the net I found this CC-BY-NC 3.0 windmill 3D model by ahedov. I downloaded the .blend file from that site, loaded it into Blender and exported it as an .OBJ file.

> *Note: If you've never used Blender you might be in for a surprise in that Blender does things differently than just about every other program you've*

> *ever used. Just be aware you might need to set aside some time to read some basic UI navigation for Blender.*
>
> *Let me also add that 3D programs in general are giant beasts with 1000s of features. They are some of the most complicated software there is. When I first learned 3D Studio Max in 1996 I read through 70% of the 600 page manual spending a few hours a day for around 3 weeks. That paid off in that when I learned Maya a few years later some of the lessons learned before were applicable to Maya. So, just be aware that if you really want to be able to use 3D software to either build 3D assets or to modify existing ones put it on your schedule and clear sometime to really go through some lessons.*

In any case I used these export options. Let's try to display it!

I started with the directional lighting example from the lights article and I combined it with the hemispherical lighting example so I ended up with one HemisphereLight and one DirectionalLight. I also removed all the GUI stuff related to adjusting the lights. I also removed the cube and sphere that were being added to the scene.

Setting up the OBJLoader

From that the first thing we need to do is include the OBJLoader loader in our script. Then to load the .OBJ file we create an instance of OBJLoader, pass it the URL of our .OBJ file, and pass in a callback that adds the loaded model to our scene.

If we run that what happens? Well it's close but we're getting errors about materials since we haven't given the scene any materials and .OBJ files don't have material parameters.

The .OBJ loader can be passed an object of name / material pairs. When it loads the .OBJ file, any material name it finds it will look for the corresponding material in the map of materials set on the loader. If it finds a material that matches by name it will use that material. If not it will use the loader's default material.

```javascript
import {OBJLoader} from 'three/addons/loaders/OBJLoader.js';
```

```javascript
```

```
{
  const objLoader = new OBJLoader();
  objLoader.load('resources/models/windmill/windmill.obj', (root) => {
    scene.add(root);
  });
}
```

Loading .MTL Materials

Sometimes .OBJ files come with a .MTL file that defines materials. In our case the exporter also created a .MTL file. .MTL format is plain ASCII so it's easy to look at. Looking at it here we can see there are 2 materials referencing 5 jpg textures but where are the texture files?

All we got was an .OBJ file and an .MTL file.

At least for this model it turns out the textures are embedded in the .blend file we downloaded. We can ask blender to export those files to by picking File->External Data->Unpack All Into Files and then choosing Write Files to Current Directory. This ends up writing the files in the same folder as the .blend file in a sub folder called textures. I copied those textures into the same folder I exported the .OBJ file to.

Now that we have the textures available we can load the .MTL file. First we need to include the MTLLoader. Then we first load the .MTL file. When it's finished loading we add the just loaded materials on to the OBJLoader itself via the setMaterials and then load the .OBJ file.

javascript

```
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
import {OBJLoader} from 'three/addons/loaders/OBJLoader.js';
import {MTLLoader} from 'three/addons/loaders/MTLLoader.js';
```

javascript

```
{
  const mtlLoader = new MTLLoader();
  mtlLoader.load('resources/models/windmill/windmill.mtl', (mtl) => {
    mtl.preload();
    objLoader.setMaterials(mtl);
    objLoader.load('resources/models/windmill/windmill.obj', (root) => {
      scene.add(root);
    });
  });
}
```

plaintext

```
# Blender MTL File: 'windmill_001.blend'
# Material Count: 2

newmtl Material
Ns 0.000000
Ka 1.000000 1.000000 1.000000
Kd 0.800000 0.800000 0.800000
Ks 0.000000 0.000000 0.000000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 1
map_Kd windmill_001_lopatky_COL.jpg
map_Bump windmill_001_lopatky_NOR.jpg

newmtl windmill
Ns 0.000000
Ka 1.000000 1.000000 1.000000
Kd 0.800000 0.800000 0.800000
Ks 0.000000 0.000000 0.000000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 1
map_Kd windmill_001_base_COL.jpg
map_Bump windmill_001_base_NOR.jpg
map_Ns windmill_001_base_SPEC.jpg
```

Fixing Missing Cloth on Blades

Note that if we spin the model around you'll see the windmill cloth disappears. We need the material on the blades to be double sided, something we went over in the article on materials. There is no easy way to fix this in the .MTL file. Off the top of my head I can think of 3 ways to fix this.

- Loop over all the materials after loading them and set them all to double sided. This solution works but ideally we only want materials that need to be double sided to be double sided because drawing double sided is slower than single sided.
- Manually set a specific material. Looking in the .MTL file there are 2 materials. One called "windmill" and the other called "Material". Through trial and error I figured out the blades use the material called "Material" so we could set that one specifically.
- Realizing that the .MTL file is limited we could just not use it and instead create materials ourselves. In this case we'd need to look up the Mesh object after loading the obj file.

Which one you pick is up to you. 1 is easiest. 3 is most flexible. 2 somewhere in between. For now I'll pick 2.

javascript

```
const mtlLoader = new MTLLoader();
 mtlLoader.load('resources/models/windmill/windmill.mtl', (mtl) => {
   mtl.preload();
   for (const material of Object.values(mtl.materials)) {
     material.side = THREE.DoubleSide;
   }
   ...
```

javascript

```
const mtlLoader = new MTLLoader();
 mtlLoader.load('resources/models/windmill/windmill.mtl', (mtl) => {
   mtl.preload();
   mtl.materials.Material.side = THREE.DoubleSide;
   ...
```

javascript

```
objLoader.load('resources/models/windmill/windmill.obj', (root) => {
   const materials = {
     Material: new THREE.MeshPhongMaterial({...}),
     windmill: new THREE.MeshPhongMaterial({...}),
   };
   root.traverse(node => {
     const material = materials[node.material?.name];
     if (material) {
       node.material = material;
     }
   })
   scene.add(root);
 });
```

Fixing Blocky Appearance with Normal Maps

And with that change you should still see the cloth on the blades when looking from behind but there's one more issue. If we zoom in close we see things are turning blocky.

What's going on? Looking at the textures there are 2 textures labelled NOR for NORmal map. And looking at them they look like normal maps. Normal maps are generally purple where as bump maps are black and white. Normal maps represent the direction of the surface where as bump maps represent the height of the surface.

Looking at the source for the MTLLoader it expects the keyword norm for normal maps so let's edit the .MTL file and now when we load it it will be using the normal maps as normal maps and we can see the back of the blades.

plaintext

```
# Blender MTL File: 'windmill_001.blend'
# Material Count: 2

newmtl Material
Ns 0.000000
Ka 1.000000 1.000000 1.000000
Kd 0.800000 0.800000 0.800000
Ks 0.000000 0.000000 0.000000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 1
map_Kd windmill_001_lopatky_COL.jpg
norm windmill_001_lopatky_NOR.jpg

newmtl windmill
Ns 0.000000
Ka 1.000000 1.000000 1.000000
Kd 0.800000 0.800000 0.800000
Ks 0.000000 0.000000 0.000000
Ke 0.000000 0.000000 0.000000
Ni 1.000000
d 1.000000
illum 1
map_Kd windmill_001_base_COL.jpg
norm windmill_001_base_NOR.jpg
map_Ns windmill_001_base_SPEC.jpg
```

Loading a Different Model

Let's load a different file. Searching the net I found this CC-BY-NC windmill 3D model made by Roger Gerzner / GERIZ.3D Art. It had a .OBJ version already available. Let's load it up (note I removed the .MTL loader for now).

Hmmm, nothing appears. What's the problem? I wonder what size the model is? We can ask THREE.js what size the model is and try to set our camera automatically.

javascript

```
objLoader.load('resources/models/windmill_2/windmill.obj', (root) => {
  scene.add(root);

  const box = new THREE.Box3().setFromObject(root);
  const boxSize = box.getSize(new THREE.Vector3()).length();
  const boxCenter = box.getCenter(new THREE.Vector3());
  console.log(boxSize);
  console.log(boxCenter);
```

plaintext

```
size 2123.6499788469982
center p {x: -0.00006103515625, y: 770.0909731090069, z: -3.313507080078125}
```

Auto-Framing the Camera

Our camera is currently only showing about 100 units with near at 0.1 and far at 100. Our ground plane is only 40 units across so basically this windmill model is so big, 2000 units, that it's surrounding our camera and all parts of it our outside our frustum.

We could manually fix that but we could also make the camera auto frame our scene. Let's try that. We can then use the box we just computed adjust the camera settings to view the entire scene. Note that there is no right answer on where to put the camera. We could be facing the scene from any direction at any altitude so we'll just have to pick something.

As we went over in the article on cameras the camera defines a frustum. That frustum is defined by the field of view (fov) and the near and far settings. We want to know given whatever field of view the camera currently has, how far away does the camera need to be so the box containing the scene fits inside the frustum assuming the frustum extended forever. In other words let's assume near is 0.00000001 and far is infinity.

Since we know the size of the box and we know the field of view we have this triangle. You can see on the left is the camera and the blue frustum is projecting out in front of it. We just computed the box that contains the windmill. We need to compute how far way the camera should be from the box so that the box appears inside the frustum.

Using basic right triangle trigonometry and SOHCAHTOA, given we know the field of view for the frustum and we know the size of the box we can compute the distance.

Based on that diagram the formula for computing distance is:

distance = halfSizeToFitOnScreen / tangent(halfFovY)

Let's translate that to code. First let's make a function that will compute distance and then move the camera that distance units from the center of the box. We'll then point the camera at the center of the box.

We pass in 2 sizes. The boxSize and the sizeToFitOnScreen. If we just passed in boxSize and used that as sizeToFitOnScreen then the math would make the box fit perfectly inside the frustum. We want a little extra space above and below so we'll pass in a slightly larger size.

javascript

```
function frameArea(sizeToFitOnScreen, boxSize, boxCenter, camera) {
  const halfSizeToFitOnScreen = sizeToFitOnScreen * 0.5;
  const halfFovY = THREE.MathUtils.degToRad(camera.fov * .5);
  const distance = halfSizeToFitOnScreen / Math.tan(halfFovY);

  // compute a unit vector that points in the direction the camera is now
  // from the center of the box
  const direction = (new THREE.Vector3()).subVectors(camera.position, boxCenter).nc

  // move the camera to a position distance units way from the center
  // in whatever direction the camera was from the center already
  camera.position.copy(direction.multiplyScalar(distance).add(boxCenter));

  // pick some near and far values for the frustum that
  // will contain the box.
  camera.near = boxSize / 100;
  camera.far = boxSize * 100;

  camera.updateProjectionMatrix();

  // point the camera to look at the center of the box
  camera.lookAt(boxCenter.x, boxCenter.y, boxCenter.z);
}
```

javascript

```
{
  const objLoader = new OBJLoader();
  objLoader.load('resources/models/windmill_2/windmill.obj', (root) => {
    scene.add(root);
    // compute the box that contains all the stuff
    // from root and below
    const box = new THREE.Box3().setFromObject(root);

    const boxSize = box.getSize(new THREE.Vector3()).length();
    const boxCenter = box.getCenter(new THREE.Vector3());

    // set the camera to frame the box
    frameArea(boxSize * 1.2, boxSize, boxCenter, camera);

    // update the Trackball controls to handle the new size
    controls.maxDistance = boxSize * 10;
    controls.target.copy(boxCenter);
    controls.update();
  });
}
```

Adjusting Camera Direction and Ground Plane

This almost works. Use the mouse to rotate the camera and you should see the windmill. The problem is the windmill is large and the box's center is at about (0, 770, 0). So, when we move the camera from where it starts (0, 10, 20) to distance units way from the center in the direction the camera is relative to the center that's moving the camera almost straight down below the windmill.

Let's change it to move sideways from the center of the box to in whatever direction the camera is from the center. All we need to do to do that is zero out the y component of the vector from the box to the camera. Then, when we normalize that vector it will become a vector parallel to the XZ plane. In other words parallel to the ground.

If you look at the bottom of the windmill you'll see a small square. That is our ground plane. It's only 40x40 units and so is way too small relative to the windmill. Since the windmill is over 2000 units big let's change the size of the ground plane to something more fitting. We also need to adjust the repeat otherwise our checkerboard will be so fine we won't even be able to see it unless we zoom way way in.

```
javascript
```

```
// compute a unit vector that points in the direction the camera is now
// in the xz plane from the center of the box
const direction = (new THREE.Vector3())
    .subVectors(camera.position, boxCenter)
    .multiply(new THREE.Vector3(1, 0, 1))
    .normalize();
```

javascript

```
const planeSize = 4000;

const loader = new THREE.TextureLoader();
const texture = loader.load('resources/images/checker.png');
texture.wrapS = THREE.RepeatWrapping;
texture.wrapT = THREE.RepeatWrapping;
texture.magFilter = THREE.NearestFilter;
const repeats = planeSize / 200;
texture.repeat.set(repeats, repeats);
```

Converting TGA Textures to JPG

Let's add the materials back. Like before there is a .MTL file that references some textures but looking at the files I quickly see an issue.

There are TARGA (.tga) files and they are giant!

THREE.js actually has a TGA loader but it's arguably wrong to use it for most use cases. If you're making a viewer where you want to allow users to view random 3D files they find on the net then maybe, just maybe, you might want to load TGA files.

One problem with TGA files are they can't be compressed well at all. TGA only supports very simple compression and looking above we can see the files are not compressed at all as the odds of them being all exactly the same size are extremely low. Further they are 12 megabytes each!!! If we used those files the user would have to download 36meg to see the windmill.

Another issue with TGA is the browser itself has no support for them so loading them is likely going to be slower than loading supported formats like .JPG and .PNG

I'm pretty sure for our purposes converting them to .JPG will be the best option. Looking inside I see they are 3 channels each, RGB, there is no alpha channel. JPG only supports 3 channels so that's a good fit. JPG also supports lossy compression so we can make the files much smaller to download.

Loading the files up they were each 2048x2048. That seemed like a waste to me but of course it depends on your use case. I made them each 1024x1024 and saved them at a 50% quality setting in Photoshop.

We went from 36meg to 0.55meg! Of course the artist might not be pleased with this compression so be sure to consult with them to discuss the tradeoffs.

Now, to use the .MTL file we need to edit it to reference the .JPG files instead of the .TGA files. Fortunately it's a simple text file so it's easy to edit.

plaintext

```
newmtl blinn1SG
Ka 0.10 0.10 0.10
Kd 0.00 0.00 0.00
Ks 0.00 0.00 0.00
Ke 0.00 0.00 0.00
Ns 0.060000
Ni 1.500000
d 1.000000
Tr 0.000000
Tf 1.000000 1.000000 1.000000
illum 2
map_Kd windmill_diffuse.jpg
map_Ks windmill_spec.jpg
map_bump windmill_normal.jpg
bump windmill_normal.jpg
```

Fixing MTL File Issues

Now that the .MTL file points to some reasonable size textures we need to load it so we'll just do like we did above, first load the materials and then set them on the OBJLoader.

Before we actually try it out I ran into some issues that rather than show a failure I'm just going to go over them.

Issue #1: The three MTLLoader creates materials that multiply the material's diffuse color by the diffuse texture map.

That's a useful feature but looking a the .MTL file above the line:

Kd 0.00 0.00 0.00

sets the diffuse color to 0. Texture map * 0 = black! It's possible the modeling tool used to make the windmill did not multiply the diffuse texture map by the diffuse color. That's why it worked for the artists that made this windmill.

To fix this we can change the line to:

Kd 1.00 1.00 1.00

since Texture Map * 1 = Texture Map.

Issue #2: The specular color is also black

The line that starts with Ks specifies the specular color. It's likely the modeling software used to make the windmill did something similar as it did with diffuse maps in that it used the specular map's color for specular highlights. Three.js uses only the red channel of a specular map as input to how much of the specular color to reflect but three still needs a specular color set.

Like above we can fix that by editing the .MTL file.

Issue #3: The windmill_normal.jpg is a normal map not a bump map.

Just like above we just need to edit the .MTL file.

javascript

```
{
  const mtlLoader = new MTLLoader();
  mtlLoader.load('resources/models/windmill_2/windmill-fixed.mtl', (mtl) => {
    mtl.preload();
    const objLoader = new OBJLoader();
    objLoader.setMaterials(mtl);
    objLoader.load('resources/models/windmill/windmill.obj', (root) => {
      root.updateMatrixWorld();
      scene.add(root);
      // compute the box that contains all the stuff
      // from root and below
      const box = new THREE.Box3().setFromObject(root);

      const boxSize = box.getSize(new THREE.Vector3()).length();
      const boxCenter = box.getCenter(new THREE.Vector3());

      // set the camera to frame the box
      frameArea(boxSize * 1.2, boxSize, boxCenter, camera);

      // update the Trackball controls to handle the new size
      controls.maxDistance = boxSize * 10;
      controls.target.copy(boxCenter);
      controls.update();
    });
  });
}
```

plaintext

```
Kd 1.00 1.00 1.00
```

plaintext

```
Ks 1.00 1.00 1.00
```

plaintext

```
norm windmill_normal.jpg
```

Common Loading Issues

Loading models often runs into these kinds of issues. Common issues include:

- Needing to know the size: Above we made the camera try to frame the scene but that's not always the appropriate thing to do. Generally the most appropriate thing to do is to make your own models or download the models, load them up in some 3D software and look at their scale and adjust if need be.
- Orientation Wrong: THREE.js is generally Y = up. Some modeling packages default to Z = up, some Y = up. Some are settable. If you run into this case where you load a model and it's on its side. You can either hack your code to rotate the model after loading (not recommended), or you can load the model into your favorite modeling package or use some command line tools to rotate the object in the orientation you need it to be just like you'd edit an image for your website rather than download it and apply code to adjust it. Blender even has options when you export to change the orientation.
- No .MTL file or wrong materials or incompatible parameters: Above we used a .MTL file above which helped us load materials but there were issues. We manually edited the .MTL file to fix. It's also common to look inside the .OBJ file to see what materials there are, or to load the .OBJ file in THREE.js and walk the scene and print out all the materials. Then, go modify the code to make custom materials and assign them where appropriate either by making a name/material pair object to pass to the loader instead of loading the .MTL file, OR, after the scene has loaded, walking the scene and fixing things.
- Textures too large: Most 3D models are made for either architecture, movies and commercials, or games. For architecture and movies no one really cares about the size of the textures since. For games people care because games have limited memory but most games run locally. Webpages though you want to load as fast as

> possible and so you need to look at the textures and try to make them as small as possible and still look good. In fact the first windmill we should arguably done something about the textures. They are currently a total of 10meg!!!

Also remember like we mentioned in the article on textures that textures take memory so a 50k JPG that expands to 4096x4096 will download fast but still take a ton of memory.

Spinning Limitation and Next Steps

The last thing I wanted to show is spinning the windmills. Unfortunately, .OBJ files have no hierarchy. That means all parts of each windmill are basically considered 1 single mesh. You can't spin the blades of the mill as they aren't separated from the rest of the building.

This is one of the main reasons why .OBJ is not really a good format. If I was to guess, the reason it's more common than other formats is because it's simple and doesn't support many features it works more often than not. Especially if you're making something still like an architectural image and there's no need to animate anything it's not a bad way to get static props into a scene.

Next up we'll try loading a gLTF scene. The gLTF format supports many more features.

Loading a .GLTF File

GUIDE

Source: https://threejs.org/manual/en/load-gltf.html

A tutorial on loading and working with .GLTF files in three.js, covering loading the file, inspecting the scene graph, animating objects, creating paths, and adding shadows.

Loading a .GLTF File

In a previous lesson we loaded an .OBJ file. If you haven't read it you might want to check it out first.

As pointed out over there the .OBJ file format is very old and fairly simple. It provides no scene graph so everything loaded is one large mesh. It was designed mostly as a simple way to pass data between 3D editors.

The gLTF format is actually a format designed from the ground up for be used for displaying graphics. 3D formats can be divided into 3 or 4 basic types.

- 3D Editor Formats: This are formats specific to a single app. .blend (Blender), .max (3d Studio Max), .mb and .ma (Maya), etc...
- Exchange formats: These are formats like .OBJ, .DAE (Collada), .FBX. They are designed to help exchange information between 3D editors. As such they are usually much larger than needed with extra info used only inside 3d editors
- App formats: These are usually specific to certain apps, usually games.
- Transmission formats: gLTF might be the first true transmission format. I suppose VRML might be considered one but VRML was actually a pretty poor format.

gLTF is designed to do some things well that all those other formats don't do:

- Be small for transmission: For example this means much of their large data, like vertices, is stored in binary. When you download a .gLTF file that data can be uploaded to the GPU with zero processing. It's ready as is. This is in contrast to say VRML, .OBJ, or .DAE where vertices are stored as text and have to be parsed. Text vertex positions can easily be 3x to 5x larger than binary.
- Be ready to render: This again is different from other formats except maybe App formats. The data in a glTF file is mean to be rendered, not edited. Data that's not important to rendering has generally been removed. Polygons have been converted to triangles. Materials have known values that are supposed to work everywhere.

gLTF was specifically designed so you should be able to download a glTF file and display it with a minimum of trouble. Let's cross our fingers that's truly the case as none of the other formats have been able to do this.

Loading a gLTF Model

I wasn't really sure what I should show. At some level loading and displaying a gLTF file is simpler than an .OBJ file. Unlike a .OBJ file materials are directly part of the format. That said I thought I should at least load one up and I think going over the issues I ran into might provide some good info.

Searching the net I found this low-poly city by antonmoek which seemed like if we're lucky might make a good example.

Starting with an example from the .OBJ article I removed the code for loading .OBJ and replaced it with code for loading .GLTF

javascript

```
const mtlLoader = new MTLLoader();
mtlLoader.loadMtl('resources/models/windmill/windmill-fixed.mtl', (mtl) => {
  mtl.preload();
  mtl.materials.Material.side = THREE.DoubleSide;
  objLoader.setMaterials(mtl);
  objLoader.load('resources/models/windmill/windmill.obj', (event) => {
    const root = event.detail.loaderRootNode;
    scene.add(root);
    ...
  });
});
```

javascript

```
{
  const gltfLoader = new GLTFLoader();
  const url = 'resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf';
  gltfLoader.load(url, (gltf) => {
    const root = gltf.scene;
    scene.add(root);
    ...
  });
}
```

javascript

```
-import {LoaderSupport} from 'three/addons/loaders/LoaderSupport.js';
-import {OBJLoader} from 'three/addons/loaders/OBJLoader.js';
-import {MTLLoader} from 'three/addons/loaders/MTLLoader.js';
+import {GLTFLoader} from 'three/addons/loaders/GLTFLoader.js';
```

Inspecting the Scene Graph

I kept the auto framing code as before. We also need to include the GLTFLoader and we can get rid of the OBJLoader. Magic! It just works, textures and all.

Next I wanted to see if I could animate the cars driving around so I needed to check if the scene had the cars as separate entities and if they were setup in a way I could use them.

I wrote some code to dump put the scenegraph to the JavaScript console. Here's the code to print out the scenegraph.

And I just called it right after loading the scene. Running that I got this listing:

javascript

```
function dumpObject(obj, lines = [], isLast = true, prefix = '') {
  const localPrefix = isLast ? '└─' : '├─';
  lines.push(${prefix}${prefix ? localPrefix : ''}${obj.name || '*no-name*'} [${obj
  const newPrefix = prefix + (isLast ? '  ' : '│ ');
  const lastNdx = obj.children.length - 1;
  obj.children.forEach((child, ndx) => {
    const isLast = ndx === lastNdx;
    dumpObject(child, lines, isLast, newPrefix);
  });
  return lines;
}
```

javascript

```
const gltfLoader = new GLTFLoader();
gltfLoader.load('resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf',
  const root = gltf.scene;
  scene.add(root);
  console.log(dumpObject(root).join('\n'));
```

text

```
OSG_Scene [Scene]
  └─RootNode_(gltf_orientation_matrix) [Object3D]
    └─RootNode_(model_correction_matrix) [Object3D]
      └─4d4100bcb1c640e69699a87140df79d7fbx [Object3D]
        └─RootNode [Object3D]
          | ...
          ├─Cars [Object3D]
          | ├─CAR_03_1 [Object3D]
          | | └─CAR_03_1_World_ap_0 [Mesh]
          | ├─CAR_03 [Object3D]
          | | └─CAR_03_World_ap_0 [Mesh]
          | ├─Car_04 [Object3D]
          | | └─Car_04_World_ap_0 [Mesh]
          | ├─CAR_03_2 [Object3D]
          | | └─CAR_03_2_World_ap_0 [Mesh]
          | ├─Car_04_1 [Object3D]
          | | └─Car_04_1_World_ap_0 [Mesh]
          | ├─Car_04_2 [Object3D]
          | | └─Car_04_2_World_ap_0 [Mesh]
          | ├─Car_04_3 [Object3D]
          | | └─Car_04_3_World_ap_0 [Mesh]
          | ├─Car_04_4 [Object3D]
          | | └─Car_04_4_World_ap_0 [Mesh]
          | ├─Car_08_4 [Object3D]
          | | └─Car_08_4_World_ap8_0 [Mesh]
          | ├─Car_08_3 [Object3D]
          | | └─Car_08_3_World_ap9_0 [Mesh]
          | ├─Car_04_1_2 [Object3D]
          | | └─Car_04_1_2_World_ap_0 [Mesh]
          | ├─Car_08_2 [Object3D]
          | | └─Car_08_2_World_ap11_0 [Mesh]
          | ├─CAR_03_1_2 [Object3D]
          | | └─CAR_03_1_2_World_ap_0 [Mesh]
          | ├─CAR_03_2_2 [Object3D]
          | | └─CAR_03_2_2_World_ap_0 [Mesh]
          | ├─Car_04_2_2 [Object3D]
          | | └─Car_04_2_2_World_ap_0 [Mesh]
          ...
```

Rotating the Cars

From that we can see all the cars happen to be under a parent called "Cars".

So as a simple test I thought I would just try rotating all the children of the "Cars" node around their Y axis.

I looked up the "Cars" node after loading the scene and saved the result.

Then in the render function we can just set the rotation of each child of cars.

And we get... Hmmm, it looks like unfortunately this scene wasn't designed to animate the cars as their origins are not setup for that purpose. The trucks are rotating in the wrong direction.

This brings up an important point which is if you're going to do something in 3D you need to plan ahead and design your assets so they have their origins in the correct places, so they are the correct scale, etc.

javascript

```
+let cars;
{
  const gltfLoader = new GLTFLoader();
  gltfLoader.load('resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf
    const root = gltf.scene;
    scene.add(root);
+    cars = root.getObjectByName('Cars');
```

javascript

```
+function render(time) {
+  time *= 0.001;  // convert to seconds

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

+  if (cars) {
+    for (const car of cars.children) {
+      car.rotation.y = time;
+    }
+  }

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

Fixing Car Orientation

Since I'm not an artist and I don't know blender that well I will hack this example. We'll take each car and parent it to another Object3D. We will then move those Object3D objects to move the cars but separately we can set the car's original Object3D to re-orient it so it's about where we really need it.

Looking back at the scene graph listing it looks like there are really only 3 types of cars, "Car_08", "CAR_03", and "Car_04". Hopefully each type of car will work with the same adjustments.

I wrote this code to go through each car, parent it to a new Object3D, parent that new Object3D to the scene, and apply some per car type settings to fix its orientation, and

add the new Object3D a cars array.

This fixes the orientation of the cars.

javascript

```
-let cars;
+const cars = [];
{
  const gltfLoader = new GLTFLoader();
  gltfLoader.load('resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf
    const root = gltf.scene;
    scene.add(root);

-    cars = root.getObjectByName('Cars');
+    const loadedCars = root.getObjectByName('Cars');
+    const fixes = [
+      { prefix: 'Car_08', rot: [Math.PI * .5, 0, Math.PI * .5], },
+      { prefix: 'CAR_03', rot: [0, Math.PI, 0], },
+      { prefix: 'Car_04', rot: [0, Math.PI, 0], },
+    ];
+
+    root.updateMatrixWorld();
+    for (const car of loadedCars.children.slice()) {
+      const fix = fixes.find(fix => car.name.startsWith(fix.prefix));
+      const obj = new THREE.Object3D();
+      car.getWorldPosition(obj.position);
+      car.position.set(0, 0, 0);
+      car.rotation.set(...fix.rot);
+      obj.add(car);
+      scene.add(obj);
+      cars.push(obj);
+    }
     ...
```

Creating a Driving Path

Now let's drive them around.

Making even a simple driving system is too much for this post but it seems instead we could just make one convoluted path that drives down all the roads and then put the cars on the path. Here's a picture from Blender about half way through building the path.

I needed a way to get the data for that path out of Blender. Fortunately I was able to select just my path and export .OBJ checking "write nurbs".

Opening the .OBJ file I was able to get a list of points which I formatted into this control points array.

THREE.js has some curve classes. The CatmullRomCurve3 seemed like it might work. The thing about that kind of curve is it tries to make a smooth curve going through the points.

In fact putting those points in directly will generate a curve like this (before image), but we want sharper corners. It seemed like if we computed some extra points we could get what we want. For each pair of points we'll compute a point 10% of the way between the 2 points and another 90% of the way between the 2 points and pass the result to CatmullRomCurve3.

This will give us a curve like this (after image).

Here's the code to make the curve. The first part of that code makes a curve. The second part of that code generates 250 points from the curve and then creates an object to display the lines made by connecting those 250 points.

Running the example I didn't see the curve. To make it visible I made it ignore the depth test and render last. And that's when I discovered it was way too small.

Checking the hierarchy in Blender I found out that the artist had scaled the node all the cars are parented to.

Scaling is bad for real time 3D apps. It causes all kinds of issues and ends up being no end of frustration when doing real time 3D. Artists often don't know this because it's so easy to scale an entire scene in a 3D editing program but if you decide to make a real time 3D app I suggest you request your artists to never scale anything. If they change the scale they should find a way to apply that scale to the vertices so that when it ends up making it to your app you can ignore scale.

And, not just scale, in this case the cars are rotated and offset by their parent, the Cars node. This will make it hard at runtime to move the cars around in world space. To be clear, in this case we want cars to drive around in world space which is why these issues are coming up. If something that is meant to be manipulated in a local space, like the moon revolving around the earth this is less of an issue.

```javascript
```

```
const controlPoints = [
  [1.118281, 5.115846, -3.681386],
  [3.948875, 5.115846, -3.641834],
  [3.960072, 5.115846, -0.240352],
  [3.985447, 5.115846, 4.585005],
  [-3.793631, 5.115846, 4.585006],
  [-3.826839, 5.115846, -14.736200],
  [-14.542292, 5.115846, -14.765865],
  [-14.520929, 5.115846, -3.627002],
  [-5.452815, 5.115846, -3.634418],
  [-5.467251, 5.115846, 4.549161],
  [-13.266233, 5.115846, 4.567083],
  [-13.250067, 5.115846, -13.499271],
  [4.081842, 5.115846, -13.435463],
  [4.125436, 5.115846, -5.334928],
  [-14.521364, 5.115846, -5.239871],
  [-14.510466, 5.115846, 5.486727],
  [5.745666, 5.115846, 5.510492],
  [5.787942, 5.115846, -14.728308],
  [-5.423720, 5.115846, -14.761919],
  [-5.373599, 5.115846, -3.704133],
  [1.004861, 5.115846, -3.641834],
];
```

javascript

```
let curve;
let curveObject;
{
  const controlPoints = [
    [1.118281, 5.115846, -3.681386],
    [3.948875, 5.115846, -3.641834],
    [3.960072, 5.115846, -0.240352],
    [3.985447, 5.115846, 4.585005],
    [-3.793631, 5.115846, 4.585006],
    [-3.826839, 5.115846, -14.736200],
    [-14.542292, 5.115846, -14.765865],
    [-14.520929, 5.115846, -3.627002],
    [-5.452815, 5.115846, -3.634418],
    [-5.467251, 5.115846, 4.549161],
    [-13.266233, 5.115846, 4.567083],
    [-13.250067, 5.115846, -13.499271],
    [4.081842, 5.115846, -13.435463],
    [4.125436, 5.115846, -5.334928],
    [-14.521364, 5.115846, -5.239871],
    [-14.510466, 5.115846, 5.486727],
    [5.745666, 5.115846, 5.510492],
    [5.787942, 5.115846, -14.728308],
    [-5.423720, 5.115846, -14.761919],
    [-5.373599, 5.115846, -3.704133],
    [1.004861, 5.115846, -3.641834],
  ];
  const p0 = new THREE.Vector3();
  const p1 = new THREE.Vector3();
  curve = new THREE.CatmullRomCurve3(
    controlPoints.map((p, ndx) => {
      p0.set(...p);
      p1.set(...controlPoints[(ndx + 1) % controlPoints.length]);
      return [
        (new THREE.Vector3()).copy(p0),
        (new THREE.Vector3()).lerpVectors(p0, p1, 0.1),
        (new THREE.Vector3()).lerpVectors(p0, p1, 0.9),
      ];
    }).flat(),
    true,
  );
  {
    const points = curve.getPoints(250);
    const geometry = new THREE.BufferGeometry().setFromPoints(points);
    const material = new THREE.LineBasicMaterial({color: 0xff0000});
    curveObject = new THREE.Line(geometry, material);
    scene.add(curveObject);
  }
}
```

javascript

```
curveObject = new THREE.Line(geometry, material);
+    material.depthTest = false;
+    curveObject.renderOrder = 1;
```

Debugging Scene Graph with Transforms

Going back to the function we wrote above to dump the scene graph, let's dump the position, rotation, and scale of each node.

And the result from running it shows us that Cars in the original scene has had its rotation and scale removed and applied to its children. That suggests either whatever exporter was used to create the .GLTF file did some special work here or more likely the artist exported a different version of the file than the corresponding .blend file, which is why things don't match.

The moral of that is I should have probably downloaded the .blend file and exported myself. Before exporting I should have inspected all the major nodes and removed any transformations.

All these nodes at the top of the hierarchy are also a waste.

Ideally the scene would consist of a single "root" node with no position, rotation, or scale. At runtime I could then pull all the children out of that root and parent them to the scene itself. There might be children of the root like "Cars" which would help me find all the cars but ideally it would also have no translation, rotation, or scale so I could re-parent the cars to the scene with the minimal amount of work.

javascript

```
+function dumpVec3(v3, precision = 3) {
+  return ${v3.x.toFixed(precision)}, ${v3.y.toFixed(precision)}, ${v3.z.toFixed(pr
+}

function dumpObject(obj, lines, isLast = true, prefix = '') {
  const localPrefix = isLast ? '└─' : '├─';
  lines.push(${prefix}${prefix ? localPrefix : ''}${obj.name || '*no-name*'} [${obj
+  const dataPrefix = obj.children.length
+     ? (isLast ? '  │ ' : '│ │ ')
+     : (isLast ? '    ' : '│   ');
+  lines.push(${prefix}${dataPrefix}  pos: ${dumpVec3(obj.position)});
+  lines.push(${prefix}${dataPrefix}  rot: ${dumpVec3(obj.rotation)});
+  lines.push(${prefix}${dataPrefix}  scl: ${dumpVec3(obj.scale)});
  const newPrefix = prefix + (isLast ? '  ' : '│ ');
  const lastNdx = obj.children.length - 1;
  obj.children.forEach((child, ndx) => {
    const isLast = ndx === lastNdx;
    dumpObject(child, lines, isLast, newPrefix);
  });
  return lines;
}
```

text

```
OSG_Scene [Scene]
  |   pos: 0.000, 0.000, 0.000
  |   rot: 0.000, 0.000, 0.000
  |   scl: 1.000, 1.000, 1.000
  └─RootNode_(gltf_orientation_matrix) [Object3D]
    |   pos: 0.000, 0.000, 0.000
    |   rot: -1.571, 0.000, 0.000
    |   scl: 1.000, 1.000, 1.000
    └─RootNode_(model_correction_matrix) [Object3D]
      |   pos: 0.000, 0.000, 0.000
      |   rot: 0.000, 0.000, 0.000
      |   scl: 1.000, 1.000, 1.000
      └─4d4100bcb1c640e69699a87140df79d7fbx [Object3D]
        |   pos: 0.000, 0.000, 0.000
        |   rot: 1.571, 0.000, 0.000
        |   scl: 1.000, 1.000, 1.000
        └─RootNode [Object3D]
          |   pos: 0.000, 0.000, 0.000
          |   rot: 0.000, 0.000, 0.000
          |   scl: 1.000, 1.000, 1.000
          ├─Cars [Object3D]
*         | |   pos: -369.069, -90.704, -920.159
*         | |   rot: 0.000, 0.000, 0.000
*         | |   scl: 1.000, 1.000, 1.000
          | ├─CAR_03_1 [Object3D]
          | | |   pos: 22.131, 14.663, -475.071
          | | |   rot: -3.142, 0.732, 3.142
          | | |   scl: 1.500, 1.500, 1.500
          | | └─CAR_03_1_World_ap_0 [Mesh]
          | |       pos: 0.000, 0.000, 0.000
          | |       rot: 0.000, 0.000, 0.000
          | |       scl: 1.000, 1.000, 1.000
```

Moving Cars Along the Path

In any case the quickest though maybe not the best fix is to just adjust the object we're using to view the curve.

Here's what I ended up with. First I adjusted the position of the curve and found values that seemed to work. I then hid it.

Then I wrote code to move the cars along the curve. For each car we pick a position from 0 to 1 along the curve and compute a point in world space using the curveObject to transform the point. We then pick another point slightly further down the curve. We set the car's orientation using lookAt and put the car at the mid point between the 2 points.

and when I ran it I found out for each type of car, their height above their origins are not consistently set and so I needed to offset each one a little.

And the result. Not bad for a few minutes work.

javascript

```
{
  const points = curve.getPoints(250);
  const geometry = new THREE.BufferGeometry().setFromPoints(points);
  const material = new THREE.LineBasicMaterial({color: 0xff0000});
  curveObject = new THREE.Line(geometry, material);
+  curveObject.scale.set(100, 100, 100);
+  curveObject.position.y = -621;
+  curveObject.visible = false;
  material.depthTest = false;
  curveObject.renderOrder = 1;
  scene.add(curveObject);
}
```

javascript

```
// create 2 Vector3s we can use for path calculations
const carPosition = new THREE.Vector3();
const carTarget = new THREE.Vector3();

function render(time) {
  ...

-  for (const car of cars) {
-    car.rotation.y = time;
-  }

+  {
+    const pathTime = time * .01;
+    const targetOffset = 0.01;
+    cars.forEach((car, ndx) => {
+      // a number between 0 and 1 to evenly space the cars
+      const u = pathTime + ndx / cars.length;
+
+      // get the first point
+      curve.getPointAt(u % 1, carPosition);
+      carPosition.applyMatrix4(curveObject.matrixWorld);
+
+      // get a second point slightly further down the curve
+      curve.getPointAt((u + targetOffset) % 1, carTarget);
+      carTarget.applyMatrix4(curveObject.matrixWorld);
+
+      // put the car at the first point (temporarily)
+      car.position.copy(carPosition);
+      // point the car the second point
+      car.lookAt(carTarget);
+
+      // put the car between the 2 points
+      car.position.lerpVectors(carPosition, carTarget, 0.5);
+    });
+  }
```

javascript

```
const loadedCars = root.getObjectByName('Cars');
const fixes = [
-  { prefix: 'Car_08', rot: [Math.PI * .5, 0, Math.PI * .5], },
-  { prefix: 'CAR_03', rot: [0, Math.PI, 0], },
-  { prefix: 'Car_04', rot: [0, Math.PI, 0], },
+  { prefix: 'Car_08', y: 0,  rot: [Math.PI * .5, 0, Math.PI * .5], },
+  { prefix: 'CAR_03', y: 33, rot: [0, Math.PI, 0], },
+  { prefix: 'Car_04', y: 40, rot: [0, Math.PI, 0], },
];

root.updateMatrixWorld();
for (const car of loadedCars.children.slice()) {
  const fix = fixes.find(fix => car.name.startsWith(fix.prefix));
  const obj = new THREE.Object3D();
  car.getWorldPosition(obj.position);
-  car.position.set(0, 0, 0);
+  car.position.set(0, fix.y, 0);
  car.rotation.set(...fix.rot);
  obj.add(car);
  scene.add(obj);
  cars.push(obj);
}
```

Adding Shadows

The last thing I wanted to do is turn on shadows.

To do this I grabbed all the GUI code from the DirectionalLight shadows example in the article on shadows and pasted it into our latest code.

Then, after loading, we need to turn on shadows on all the objects.

I then spent nearly 4 hours trying to figure out why the shadow helpers were not working. It was because I forgot to enable shadows with renderer.shadowMap.enabled = true.

I then adjusted the values until our DirectionLight's shadow camera had a frustum that covered the entire scene. These are the settings I ended up with.

and I set the background color to light blue.

And ... shadows!

```
javascript
```

```
{
  const gltfLoader = new GLTFLoader();
  gltfLoader.load('resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf
    const root = gltf.scene;
    scene.add(root);

+    root.traverse((obj) => {
+      if (obj.castShadow !== undefined) {
+        obj.castShadow = true;
+        obj.receiveShadow = true;
+      }
+    });
```

javascript

```
renderer.shadowMap.enabled = true;
```

javascript

```
{
  const color = 0xFFFFFF;
  const intensity = 1;
  const light = new THREE.DirectionalLight(color, intensity);
+  light.castShadow = true;
*  light.position.set(-250, 800, -850);
*  light.target.position.set(-550, 40, -450);

+  light.shadow.bias = -0.004;
+  light.shadow.mapSize.width = 2048;
+  light.shadow.mapSize.height = 2048;

  scene.add(light);
  scene.add(light.target);
+  const cam = light.shadow.camera;
+  cam.near = 1;
+  cam.far = 2000;
+  cam.left = -1500;
+  cam.right = 1500;
+  cam.top = 1500;
+  cam.bottom = -1500;
...
```

javascript

```
const scene = new THREE.Scene();
-scene.background = new THREE.Color('black');
+scene.background = new THREE.Color('#DEFEFF');
```

Conclusion

I hope walking through this project was useful and showed some good examples of working though some of the issues of loading a file with a scenegraph.

One interesting thing is that comparing the .blend file to the .gltf file, the .blend file has several lights but they are not lights after being loaded into the scene. A .GLTF file is just a JSON file so you can easily look inside. It consists of several arrays of things and each item in an array is referenced by index else where. While there are extensions in the works they point to a problem with almost all 3d formats. They can never cover every case.

There is always a need for more data. For example we manually exported a path for the cars to follow. Ideally that info could have been in the .GLTF file but to do that we'd need to write our own exporter and some how mark nodes for how we want them exported or use a naming scheme or something along those lines to get data from whatever tool we're using to create the data into our app.

All of that is left as an exercise to the reader.

Advanced Rendering

How to use Post Processing

GUIDE

Source: https://threejs.org/manual/en/how-to-use-post-processing.html

A guide explaining how to use three.js's post-processing pipeline via EffectComposer to apply graphical effects such as Depth-Of-Field, Bloom, Film Grain, and Anti-aliasing to rendered scenes.

How to use Post Processing

Many three.js applications render their 3D objects directly to the screen. Sometimes, however, you want to apply one or more graphical effects like Depth-Of-Field, Bloom, Film Grain or various types of Anti-aliasing. Post-processing is a widely used approach to implement such effects. First, the scene is rendered to a render target which represents a buffer in the video card's memory. In the next step one or more post-processing passes apply filters and effects to the image buffer before it is eventually rendered to the screen.

three.js provides a complete post-processing solution via `EffectComposer` to implement such a workflow.

Workflow

The first step in the process is to import all necessary files from the examples directory. The guide assumes you are using the official npm package of three.js. For our basic demo in this guide we need the following files.

After all files are successfully imported, we can create our composer by passing in an instance of `WebGLRenderer`.

When using a composer, it's necessary to change the application's animation loop. Instead of calling the render method of `WebGLRenderer`, we now use the respective counterpart of `EffectComposer`.

Our composer is now ready so it's possible to configure the chain of post-processing passes. These passes are responsible for creating the final visual output of the application. They are processed in order of their addition/insertion. In our example, the instance of `RenderPass` is executed first, then the instance of `GlitchPass` and finally `OutputPass`. The last enabled pass in the chain is automatically rendered to the screen. The setup of the passes looks like so:

`RenderPass` is normally placed at the beginning of the chain in order to provide the rendered scene as an input for the next post-processing step. In our case, `GlitchPass` is going to use these image data to apply a wild glitch effect. `OutputPass` is usually the last pass in the chain which performs sRGB color space conversion and tone mapping. Check out this live example to see it in action.

```javascript
import { EffectComposer } from 'three/addons/postprocessing/EffectComposer.js';
import { RenderPass } from 'three/addons/postprocessing/RenderPass.js';
import { GlitchPass } from 'three/addons/postprocessing/GlitchPass.js';
import { OutputPass } from 'three/addons/postprocessing/OutputPass.js';
```

javascript

```
const composer = new EffectComposer( renderer );
```

javascript

```
function animate() {

  requestAnimationFrame( animate );

  composer.render();

}
```

javascript

```
const renderPass = new RenderPass( scene, camera );
composer.addPass( renderPass );

const glitchPass = new GlitchPass();
composer.addPass( glitchPass );

const outputPass = new OutputPass();
composer.addPass( outputPass );
```

Built-in Passes

You can use a wide range of pre-defined post-processing passes provided by the engine. They are located in the postprocessing directory.

Custom Passes

Sometimes you want to write a custom post-processing shader and include it into the chain of post-processing passes. For this scenario, you can utilize `ShaderPass`. After importing the file and your custom shader, you can use the following code to setup the pass.

The repository provides a file called CopyShader which is a good starting code for your own custom shader. `CopyShader` just copies the image contents of the `EffectComposer`'s read buffer to its write buffer without applying any effects.

javascript

```
import { ShaderPass } from 'three/addons/postprocessing/ShaderPass.js';
import { LuminosityShader } from 'three/addons/shaders/LuminosityShader.js';

// later in your init routine

const luminosityPass = new ShaderPass( LuminosityShader );
composer.addPass( luminosityPass );
```

Post Processing

GUIDE

Source: https://threejs.org/manual/en/post-processing.html

An introduction to post processing in THREE.js, covering the EffectComposer pipeline, built-in passes such as RenderPass, BloomPass, FilmPass, and OutputPass, runtime parameter adjustment via uniforms, and creating custom shader passes with ShaderPass.

Overview

Post processing generally refers to applying some kind of effect or filter to a 2D image. In the case of THREE.js we have a scene with a bunch of meshes in it. We render that scene into a 2D image. Normally that image is rendered directly into the canvas and displayed in the browser but instead we can render it to a render target and then apply some post processing effects to the result before drawing it to the canvas. It's called post processing because it happens after (post) the main scene processing.

Examples of post processing are Instagram like filters, Photoshop filters, etc...

THREE.js has some example classes to help setup a post processing pipeline. The way it works is you create an EffectComposer and to it you add multiple Pass objects. You then call EffectComposer.render and it renders your scene to a render target and then applies each Pass.

Each Pass can be some post processing effect like adding a vignette, blurring, applying a bloom, applying film grain, adjusting the hue, saturation, contrast, etc... and finally rendering the result to the canvas.

It's a little bit important to understand how EffectComposer functions. It creates two render targets. Let's call them rtA and rtB.

Then, you call EffectComposer.addPass to add each pass in the order you want to apply them. The passes are then applied something like this. First the scene you

passed into RenderPass is rendered to rtA, then rtA is passed to the next pass, whatever it is. That pass uses rtA as input to do whatever it does and writes the results to rtB. rtB is then passed to the next pass which uses rtB as input and writes back to rtA. This continues through all the passes.

Each Pass has 4 basic options

enabled

Whether or not to use this pass

needsSwap

Whether or not to swap rtA and rtB after finishing this pass

clear

Whether or not to clear before rendering this pass

renderToScreen

Whether or not to render to the canvas instead the current destination render target. In most use cases you do not set this flag explicitly since the last pass in the pass chain is automatically rendered to screen.

Basic Example

Let's put together a basic example. We'll start with the example from the article on responsiveness.

To that first we create an EffectComposer.

Then as the first pass we add a RenderPass that will render our scene with our camera into the first render target.

Next we add a BloomPass. A BloomPass renders its input to a generally smaller render target and blurs the result. It then adds that blurred result on top of the original input. This makes the scene bloom.

Next we had a FilmPass that draws noise and scanlines on top of its input.

Finally we had a OutputPass which performs color space conversion to sRGB and optional tone mapping. This pass is usually the last pass of the pass chain.

To use these classes we need to import a bunch of scripts.

For pretty much any post processing EffectComposer.js, RenderPass.js and OutputPass.js are required.

The last things we need to do are to use EffectComposer.render instead of WebGLRenderer.render and to tell the EffectComposer to match the size of the canvas.

EffectComposer.render takes a deltaTime which is the time in seconds since the last frame was rendered. It passes this to the various effects in case any of them are animated. In this case the FilmPass is animated.

javascript

```
const composer = new EffectComposer(renderer);
```

javascript

```
composer.addPass(new RenderPass(scene, camera));
```

javascript

```
const bloomPass = new BloomPass(
    1,    // strength
    25,   // kernel size
    4,    // sigma ?
    256,  // blur render target resolution
);
composer.addPass(bloomPass);
```

javascript

```
const filmPass = new FilmPass(
    0.5,   // intensity
    false,  // grayscale
);
composer.addPass(filmPass);
```

javascript

```
const outputPass = new OutputPass();
composer.addPass(outputPass);
```

javascript

```
import {EffectComposer} from 'three/addons/postprocessing/EffectComposer.js';
import {RenderPass} from 'three/addons/postprocessing/RenderPass.js';
import {BloomPass} from 'three/addons/postprocessing/BloomPass.js';
import {FilmPass} from 'three/addons/postprocessing/FilmPass.js';
import {OutputPass} from 'three/addons/postprocessing/OutputPass.js';
```

javascript

```
-function render(now) {
-  time *= 0.001;
+let then = 0;
+function render(now) {
+  now *= 0.001;  // convert to seconds
+  const deltaTime = now - then;
+  then = now;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
+    composer.setSize(canvas.width, canvas.height);
  }

  cubes.forEach((cube, ndx) => {
    const speed = 1 + ndx * .1;
-    const rot = time * speed;
+    const rot = now * speed;
    cube.rotation.x = rot;
    cube.rotation.y = rot;
  });

-  renderer.render(scene, camera);
+  composer.render(deltaTime);

  requestAnimationFrame(render);
}
```

Runtime Parameter Adjustment

To change effect parameters at runtime usually requires setting uniform values. Let's add a gui to adjust some of the parameters. Figuring out which values you can easily adjust and how to adjust them requires digging through the code for that effect.

Looking inside BloomPass.js I found this line:

So we can set the strength by setting

Similarly looking in FilmPass.js I found these lines:

So which makes it pretty clear how to set them.

Let's make a quick GUI to set those values

and

and now we can adjust those settings

```javascript
this.combineUniforms[ 'strength' ].value = strength;
```

```javascript
bloomPass.combineUniforms.strength.value = someValue;
```

```javascript
this.uniforms.intensity.value = intensity;
this.uniforms.grayscale.value = grayscale;
```

```javascript
import {GUI} from 'three/addons/libs/lil-gui.module.min.js';
```

```javascript
const gui = new GUI();
{
  const folder = gui.addFolder('BloomPass');
  folder.add(bloomPass.combineUniforms.strength, 'value', 0, 2).name('strength');
  folder.open();
}
{
  const folder = gui.addFolder('FilmPass');
  folder.add(filmPass.uniforms.grayscale, 'value').name('grayscale');
  folder.add(filmPass.uniforms.intensity, 'value', 0, 1).name('intensity');
  folder.open();
}
```

Custom Shaders

That was a small step to making our own effect.

Post processing effects use shaders. Shaders are written in a language called GLSL (Graphics Library Shading Language). Going over the entire language is way too large

a topic for these articles. A few resources to get start from would be maybe this article and maybe the Book of Shaders.

I think an example to get you started would be helpful though so let's make a simple GLSL post processing shader. We'll make one that lets us multiply the image by a color.

For post processing THREE.js provides a useful helper called the ShaderPass. It takes an object with info defining a vertex shader, a fragment shader, and the default inputs. It will handling setting up which texture to read from to get the previous pass's results and where to render to, either one of the EffectComposer's render target or the canvas.

Here's a simple post processing shader that multiplies the previous pass's result by a color.

Above tDiffuse is the name that ShaderPass uses to pass in the previous pass's result texture so we pretty much always need that. We then declare color as a THREE.js Color.

Next we need a vertex shader. For post processing the vertex shader shown here is pretty much standard and rarely needs to be changed. Without going into too many details (see articles linked above) the variables uv, projectionMatrix, modelViewMatrix and position are all magically added by THREE.js.

Finally we create a fragment shader. In it we get a pixel color from the previous pass with this line

we multiply it by our color and set gl_FragColor to the result

Adding some simple GUI to set the 3 values of the color

Gives us a simple postprocessing effect that multiplies by a color.

```
javascript
```

```
const colorShader = {
  uniforms: {
    tDiffuse: { value: null },
    color:    { value: new THREE.Color(0x88CCFF) },
  },
  vertexShader:
    varying vec2 vUv;
    void main() {
      vUv = uv;
      gl_Position = projectionMatrix * modelViewMatrix * vec4(position, 1);
    }
  ,
  fragmentShader:
    varying vec2 vUv;
    uniform sampler2D tDiffuse;
    uniform vec3 color;
    void main() {
      vec4 previousPassColor = texture2D(tDiffuse, vUv);
      gl_FragColor = vec4(
          previousPassColor.rgb * color,
          previousPassColor.a);
    }
  ,
};
```

glsl

```
vec4 previousPassColor = texture2D(tDiffuse, vUv);
```

glsl

```
gl_FragColor = vec4(
    previousPassColor.rgb * color,
    previousPassColor.a);
```

javascript

```
const gui = new GUI();
gui.add(colorPass.uniforms.color.value, 'r', 0, 4).name('red');
gui.add(colorPass.uniforms.color.value, 'g', 0, 4).name('green');
gui.add(colorPass.uniforms.color.value, 'b', 0, 4).name('blue');
```

Additional Resources

As mentioned about all the details of how to write GLSL and custom shaders is too much for these articles. If you really want to know how WebGL itself works then check out these articles. Another great resources is just to read through the existing post processing shaders in the THREE.js repo. Some are more complicated than others but if you start with the smaller ones you can hopefully get an idea of how they work.

Most of the post processing effects in the THREE.js repo are unfortunately undocumented so to use them you'll have to read through the examples or the code for the effects themselves. Hopefully these simple example and the article on render targets provide enough context to get started.

Render Targets

GUIDE

Source: https://threejs.org/manual/en/rendertargets.html

A tutorial on using WebGLRenderTarget in three.js to render a scene into a texture, then apply that texture to other objects. Covers creation, rendering setup, use cases, depth/stencil buffer options, and dynamic resizing.

Introduction

A render target in three.js is basically a texture you can render to. After you render to it you can use that texture like any other texture.

Let's make a simple example. We'll start with an example from the article on responsiveness.

Rendering to a render target is almost exactly the same as normal rendering. First we create a WebGLRenderTarget.

```javascript
const rtWidth = 512;
const rtHeight = 512;
const renderTarget = new THREE.WebGLRenderTarget(rtWidth, rtHeight);
```

Setting up a Camera and Scene for the Render Target

Then we need a Camera and a Scene.

Notice we set the aspect to the aspect for the render target, not the canvas. The correct aspect to use depends on what we are rendering for. In this case we'll use the render target's texture on the side of a cube. Since faces of the cube are square we want an aspect of 1.0.

```javascript
```

```
const rtFov = 75;
const rtAspect = rtWidth / rtHeight;
const rtNear = 0.1;
const rtFar = 5;
const rtCamera = new THREE.PerspectiveCamera(rtFov, rtAspect, rtNear, rtFar);
rtCamera.position.z = 2;

const rtScene = new THREE.Scene();
rtScene.background = new THREE.Color('red');
```

Adding Objects to the Render Target Scene

We fill the scene with stuff. In this case we're using the light and the 3 cubes from the previous article.

javascript

```
{
  const color = 0xFFFFFF;
  const intensity = 1;
  const light = new THREE.DirectionalLight(color, intensity);
  light.position.set(-1, 2, 4);
*  rtScene.add(light);
}

const boxWidth = 1;
const boxHeight = 1;
const boxDepth = 1;
const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);

function makeInstance(geometry, color, x) {
  const material = new THREE.MeshPhongMaterial({color});

  const cube = new THREE.Mesh(geometry, material);
*  rtScene.add(cube);

  cube.position.x = x;

  return cube;
}

*const rtCubes = [
  makeInstance(geometry, 0x44aa88,  0),
  makeInstance(geometry, 0x8844aa, -2),
  makeInstance(geometry, 0xaa8844,  2),
];
```

Using the Render Target as a Texture

The Scene and Camera from the previous article are still there. We'll use them to render to the canvas. We just need to add stuff to render.

Let's add a cube that uses the render target's texture.

```javascript
const material = new THREE.MeshPhongMaterial({
  map: renderTarget.texture,
});
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
```

Rendering in the Animation Loop

Now at render time first we render the render target scene to the render target.

Then we render the scene with the single cube that is using the render target's texture to the canvas.

And voilà.

```javascript
function render(time) {
  time *= 0.001;

  ...

  // rotate all the cubes in the render target scene
  rtCubes.forEach((cube, ndx) => {
    const speed = 1 + ndx * .1;
    const rot = time * speed;
    cube.rotation.x = rot;
    cube.rotation.y = rot;
  });

  // draw render target scene to render target
  renderer.setRenderTarget(renderTarget);
  renderer.render(rtScene, rtCamera);
  renderer.setRenderTarget(null);
```

```javascript
// rotate the cube in the scene
  cube.rotation.x = time;
  cube.rotation.y = time * 1.1;

  // render the scene to the canvas
  renderer.render(scene, camera);
```

Result

click here to open in a separate window

The cube is red because we set the `background` of the `rtScene` to red so the render target's texture is being cleared to red.

Common Uses for Render Targets

Render targets are used for all kinds of things. Shadows use render targets. Picking can use a render target. Various kinds of post processing effects require render targets. Rendering a rear view mirror in a car or a live view on a monitor inside a 3D scene might use a render target.

Notes on WebGLRenderTarget

A few notes about using WebGLRenderTarget.

Disabling Depth and Stencil Buffers

By default WebGLRenderTarget creates 2 textures. A color texture and a depth/stencil texture. If you don't need the depth or stencil textures you can request to not create them by passing in options. Example:

javascript

```
const rt = new THREE.WebGLRenderTarget(width, height, {
    depthBuffer: false,
    stencilBuffer: false,
  });
```

Resizing a Render Target

You might need to change the size of a render target

In the example above we make a render target of a fixed size, 512x512. For things like post processing you generally need to make a render target the same size as your canvas. In our code that would mean when we change the canvas size we would also update both the render target size and the camera we're using when rendering to the render target. Example:

javascript

```
function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();

+    renderTarget.setSize(canvas.width, canvas.height);
+    rtCamera.aspect = camera.aspect;
+    rtCamera.updateProjectionMatrix();
}
```

Custom BufferGeometry

GUIDE

Source: https://threejs.org/manual/en/custom-buffergeometry.html

A guide on how to use BufferGeometry directly in three.js to create custom geometry, including building a cube from parallel arrays, using indices to reduce duplication, working with TypedArrays, and dynamically updating vertex data.

Custom BufferGeometry

BufferGeometry is three.js's way of representing all geometry. A BufferGeometry is essentially a collection of named BufferAttributes. Each BufferAttribute represents an array of one type of data: positions, normals, colors, uv, etc... Together, the named BufferAttributes represent parallel arrays of all the data for each vertex.

Above you can see we have 4 attributes: position, normal, color, uv. They represent parallel arrays which means that the Nth set of data in each attribute belongs to the same vertex. The vertex at index = 4 is highlighted to show that the parallel data across all attributes defines one vertex.

This brings up a point, here's a diagram of a cube with one corner highlighted.

Thinking about it that single corner needs a different normal for each face of the cube. A normal is info about which direction something faces. In the diagram the normals are presented by the arrows around the corner vertex showing that each face that shares that vertex position needs a normal that points in a different direction.

That corner needs different UVs for each face as well. UVs are texture coordinates that specify which part of a texture being drawn on a triangle corresponds to that vertex position. You can see the green face needs that vertex to have a UV that corresponds

to the top right corner of the F texture, the blue face needs a UV that corresponds to the top left corner of the F texture, and the red face needs a UV that corresponds to the bottom left corner of the F texture.

A single vertex is the combination of all of its parts. If a vertex needs any part to be different then it must be a different vertex.

As a simple example let's make a cube using BufferGeometry. A cube is interesting because it appears to share vertices at the corners but really does not. For our example we'll list out all the vertices with all their data and then convert that data into parallel arrays and finally use those to make BufferAttributes and add them to a BufferGeometry.

Listing Cube Vertices

We start with a list of all the data needed for the cube. Remember again that if a vertex has any unique parts it has to be a separate vertex. As such to make a cube requires 36 vertices. 2 triangles per face, 3 vertices per triangle, 6 faces = 36 vertices.

```javascript
```

```
const vertices = [
  // front
  { pos: [-1, -1,  1], norm: [ 0,  0,  1], uv: [0, 0], },
  { pos: [ 1, -1,  1], norm: [ 0,  0,  1], uv: [1, 0], },
  { pos: [-1,  1,  1], norm: [ 0,  0,  1], uv: [0, 1], },

  { pos: [-1,  1,  1], norm: [ 0,  0,  1], uv: [0, 1], },
  { pos: [ 1, -1,  1], norm: [ 0,  0,  1], uv: [1, 0], },
  { pos: [ 1,  1,  1], norm: [ 0,  0,  1], uv: [1, 1], },
  // right
  { pos: [ 1, -1,  1], norm: [ 1,  0,  0], uv: [0, 0], },
  { pos: [ 1, -1, -1], norm: [ 1,  0,  0], uv: [1, 0], },
  { pos: [ 1,  1,  1], norm: [ 1,  0,  0], uv: [0, 1], },

  { pos: [ 1,  1,  1], norm: [ 1,  0,  0], uv: [0, 1], },
  { pos: [ 1, -1, -1], norm: [ 1,  0,  0], uv: [1, 0], },
  { pos: [ 1,  1, -1], norm: [ 1,  0,  0], uv: [1, 1], },
  // back
  { pos: [ 1, -1, -1], norm: [ 0,  0, -1], uv: [0, 0], },
  { pos: [-1, -1, -1], norm: [ 0,  0, -1], uv: [1, 0], },
  { pos: [ 1,  1, -1], norm: [ 0,  0, -1], uv: [0, 1], },

  { pos: [ 1,  1, -1], norm: [ 0,  0, -1], uv: [0, 1], },
  { pos: [-1, -1, -1], norm: [ 0,  0, -1], uv: [1, 0], },
  { pos: [-1,  1, -1], norm: [ 0,  0, -1], uv: [1, 1], },
  // left
  { pos: [-1, -1, -1], norm: [-1,  0,  0], uv: [0, 0], },
  { pos: [-1, -1,  1], norm: [-1,  0,  0], uv: [1, 0], },
  { pos: [-1,  1, -1], norm: [-1,  0,  0], uv: [0, 1], },

  { pos: [-1,  1, -1], norm: [-1,  0,  0], uv: [0, 1], },
  { pos: [-1, -1,  1], norm: [-1,  0,  0], uv: [1, 0], },
  { pos: [-1,  1,  1], norm: [-1,  0,  0], uv: [1, 1], },
  // top
  { pos: [ 1,  1, -1], norm: [ 0,  1,  0], uv: [0, 0], },
  { pos: [-1,  1, -1], norm: [ 0,  1,  0], uv: [1, 0], },
  { pos: [ 1,  1,  1], norm: [ 0,  1,  0], uv: [0, 1], },

  { pos: [ 1,  1,  1], norm: [ 0,  1,  0], uv: [0, 1], },
  { pos: [-1,  1, -1], norm: [ 0,  1,  0], uv: [1, 0], },
  { pos: [-1,  1,  1], norm: [ 0,  1,  0], uv: [1, 1], },
  // bottom
  { pos: [ 1, -1,  1], norm: [ 0, -1,  0], uv: [0, 0], },
  { pos: [-1, -1,  1], norm: [ 0, -1,  0], uv: [1, 0], },
  { pos: [ 1, -1, -1], norm: [ 0, -1,  0], uv: [0, 1], },

  { pos: [ 1, -1, -1], norm: [ 0, -1,  0], uv: [0, 1], },
  { pos: [-1, -1,  1], norm: [ 0, -1,  0], uv: [1, 0], },
  { pos: [-1, -1, -1], norm: [ 0, -1,  0], uv: [1, 1], },
];
```

Converting to Parallel Arrays

We can then translate all of that into 3 parallel arrays

```
javascript
```

```
const positions = [];
const normals = [];
const uvs = [];
for (const vertex of vertices) {
  positions.push(...vertex.pos);
  normals.push(...vertex.norm);
  uvs.push(...vertex.uv);
}
```

Creating BufferAttributes

Finally we can create a BufferGeometry and then a BufferAttribute for each array and add it to the BufferGeometry.

Note that the names are significant. You must name your attributes the names that match what three.js expects (unless you are creating a custom shader). In this case position, normal, and uv. If you want vertex colors then name your attribute color.

Above we created 3 JavaScript native arrays, positions, normals and uvs. We then convert those into TypedArrays of type Float32Array. A BufferAttribute requires a TypedArray not a native array. A BufferAttribute also requires you to tell it how many components there are per vertex. For the positions and normals we have 3 components per vertex, x, y, and z. For the UVs we have 2, u and v.

javascript

```
const geometry = new THREE.BufferGeometry();
  const positionNumComponents = 3;
  const normalNumComponents = 3;
  const uvNumComponents = 2;
  geometry.setAttribute(
      'position',
      new THREE.BufferAttribute(new Float32Array(positions), positionNumComponents)
  geometry.setAttribute(
      'normal',
      new THREE.BufferAttribute(new Float32Array(normals), normalNumComponents));
  geometry.setAttribute(
      'uv',
      new THREE.BufferAttribute(new Float32Array(uvs), uvNumComponents));
```

Using Indices

That's a lot of data. A small thing we can do is use indices to reference the vertices. Looking back at our cube data, each face is made from 2 triangles with 3 vertices each, 6 vertices total, but 2 of those vertices are exactly the same; The same position, the same normal, and the same uv. So, we can remove the matching vertices and then reference them by index. First we remove the matching vertices.

So now we have 24 unique vertices. Then we specify 36 indices for the 36 vertices we need drawn to make 12 triangles by calling BufferGeometry.setIndex with an array of indices.

javascript

```javascript
const vertices = [
  // front
  { pos: [-1, -1,  1], norm: [ 0,  0,  1], uv: [0, 0], }, // 0
  { pos: [ 1, -1,  1], norm: [ 0,  0,  1], uv: [1, 0], }, // 1
  { pos: [-1,  1,  1], norm: [ 0,  0,  1], uv: [0, 1], }, // 2

  { pos: [ 1,  1,  1], norm: [ 0,  0,  1], uv: [1, 1], }, // 3
  // right
  { pos: [ 1, -1,  1], norm: [ 1,  0,  0], uv: [0, 0], }, // 4
  { pos: [ 1, -1, -1], norm: [ 1,  0,  0], uv: [1, 0], }, // 5

  { pos: [ 1,  1,  1], norm: [ 1,  0,  0], uv: [0, 1], }, // 6
  { pos: [ 1,  1, -1], norm: [ 1,  0,  0], uv: [1, 1], }, // 7
  // back
  { pos: [ 1, -1, -1], norm: [ 0,  0, -1], uv: [0, 0], }, // 8
  { pos: [-1, -1, -1], norm: [ 0,  0, -1], uv: [1, 0], }, // 9

  { pos: [ 1,  1, -1], norm: [ 0,  0, -1], uv: [0, 1], }, // 10
  { pos: [-1,  1, -1], norm: [ 0,  0, -1], uv: [1, 1], }, // 11
  // left
  { pos: [-1, -1, -1], norm: [-1,  0,  0], uv: [0, 0], }, // 12
  { pos: [-1, -1,  1], norm: [-1,  0,  0], uv: [1, 0], }, // 13

  { pos: [-1,  1, -1], norm: [-1,  0,  0], uv: [0, 1], }, // 14
  { pos: [-1,  1,  1], norm: [-1,  0,  0], uv: [1, 1], }, // 15
  // top
  { pos: [ 1,  1, -1], norm: [ 0,  1,  0], uv: [0, 0], }, // 16
  { pos: [-1,  1, -1], norm: [ 0,  1,  0], uv: [1, 0], }, // 17

  { pos: [ 1,  1,  1], norm: [ 0,  1,  0], uv: [0, 1], }, // 18
  { pos: [-1,  1,  1], norm: [ 0,  1,  0], uv: [1, 1], }, // 19
  // bottom
  { pos: [ 1, -1,  1], norm: [ 0, -1,  0], uv: [0, 0], }, // 20
  { pos: [-1, -1,  1], norm: [ 0, -1,  0], uv: [1, 0], }, // 21

  { pos: [ 1, -1, -1], norm: [ 0, -1,  0], uv: [0, 1], }, // 22
  { pos: [-1, -1, -1], norm: [ 0, -1,  0], uv: [1, 1], }, // 23
];
```

javascript

```
geometry.setAttribute(
    'position',
    new THREE.BufferAttribute(positions, positionNumComponents));
geometry.setAttribute(
    'normal',
    new THREE.BufferAttribute(normals, normalNumComponents));
geometry.setAttribute(
    'uv',
    new THREE.BufferAttribute(uvs, uvNumComponents));

geometry.setIndex([
   0,  1,  2,   2,  1,  3,  // front
   4,  5,  6,   6,  5,  7,  // right
   8,  9, 10,  10,  9, 11,  // back
  12, 13, 14,  14, 13, 15,  // left
  16, 17, 18,  18, 17, 19,  // top
  20, 21, 22,  22, 21, 23,  // bottom
]);
```

Computing Normals Automatically

BufferGeometry has a computeVertexNormals method for computing normals if you are not supplying them. Unfortunately, since positions can not be shared if any other part of a vertex is different, the results of calling computeVertexNormals will generate seams if your geometry is supposed to connect to itself like a sphere or a cylinder.

For the cylinder above the normals were created using computeVertexNormals. If you look closely there is a seam on the cylinder. This is because there is no way to share the vertices at the start and end of the cylinder since they require different UVs so the function to compute them has no idea those are the same vertices to smooth over them. Just a small thing to be aware of. The solution is to supply your own normals.

Using TypedArrays

We can also use TypedArrays from the start instead of native JavaScript arrays. The disadvantage to TypedArrays is you must specify their size up front. Of course that's not that large of a burden but with native arrays we can just push values onto them and look at what size they end up by checking their length at the end. With TypedArrays there is no push function so we need to do our own bookkeeping when adding values to them.

In this example knowing the length up front is pretty easy since we're using a big block of static data to start.

```
javascript
```

```
const numVertices = vertices.length;
const positionNumComponents = 3;
const normalNumComponents = 3;
const uvNumComponents = 2;
const positions = new Float32Array(numVertices * positionNumComponents);
const normals = new Float32Array(numVertices * normalNumComponents);
const uvs = new Float32Array(numVertices * uvNumComponents);
let posNdx = 0;
let nrmNdx = 0;
let uvNdx = 0;
for (const vertex of vertices) {
  positions.set(vertex.pos, posNdx);
  normals.set(vertex.norm, nrmNdx);
  uvs.set(vertex.uv, uvNdx);
  posNdx += positionNumComponents;
  nrmNdx += normalNumComponents;
  uvNdx += uvNumComponents;
}

geometry.setAttribute(
    'position',
    new THREE.BufferAttribute(positions, positionNumComponents));
geometry.setAttribute(
    'normal',
    new THREE.BufferAttribute(normals, normalNumComponents));
geometry.setAttribute(
    'uv',
    new THREE.BufferAttribute(uvs, uvNumComponents));

geometry.setIndex([
   0,  1,  2,   2,  1,  3,  // front
   4,  5,  6,   6,  5,  7,  // right
   8,  9, 10,  10,  9, 11,  // back
  12, 13, 14,  14, 13, 15,  // left
  16, 17, 18,  18, 17, 19,  // top
  20, 21, 22,  22, 21, 23,  // bottom
]);
```

Dynamic Updates

A good reason to use typedarrays is if you want to dynamically update any part of the vertices.

I couldn't think of a really good example of dynamically updating the vertices so I decided to make a sphere and move each quad in and out from the center. Hopefully it's a useful example.

Here's the code to generate positions and indices for a sphere. The code is sharing vertices within a quad but it's not sharing vertices between quads because we want to be able to move each quad separately.

Because I'm lazy I used a small hierarchy of 3 Object3D objects to compute sphere points. How this works is explained in the article on optimizing lots of objects.

javascript

```
function makeSpherePositions(segmentsAround, segmentsDown) {
  const numVertices = segmentsAround * segmentsDown * 6;
  const numComponents = 3;
  const positions = new Float32Array(numVertices * numComponents);
  const indices = [];

  const longHelper = new THREE.Object3D();
  const latHelper = new THREE.Object3D();
  const pointHelper = new THREE.Object3D();
  longHelper.add(latHelper);
  latHelper.add(pointHelper);
  pointHelper.position.z = 1;
  const temp = new THREE.Vector3();

  function getPoint(lat, long) {
    latHelper.rotation.x = lat;
    longHelper.rotation.y = long;
    longHelper.updateMatrixWorld(true);
    return pointHelper.getWorldPosition(temp).toArray();
  }

  let posNdx = 0;
  let ndx = 0;
  for (let down = 0; down < segmentsDown; ++down) {
    const v0 = down / segmentsDown;
    const v1 = (down + 1) / segmentsDown;
    const lat0 = (v0 - 0.5) * Math.PI;
    const lat1 = (v1 - 0.5) * Math.PI;

    for (let across = 0; across < segmentsAround; ++across) {
      const u0 = across / segmentsAround;
      const u1 = (across + 1) / segmentsAround;
      const long0 = u0 * Math.PI * 2;
      const long1 = u1 * Math.PI * 2;

      positions.set(getPoint(lat0, long0), posNdx);  posNdx += numComponents;
      positions.set(getPoint(lat1, long0), posNdx);  posNdx += numComponents;
      positions.set(getPoint(lat0, long1), posNdx);  posNdx += numComponents;
      positions.set(getPoint(lat1, long1), posNdx);  posNdx += numComponents;

      indices.push(
        ndx, ndx + 1, ndx + 2,
        ndx + 2, ndx + 1, ndx + 3,
      );
      ndx += 4;
    }
  }
  return {positions, indices};
}
```

javascript

```
const segmentsAround = 24;
const segmentsDown = 16;
const {positions, indices} = makeSpherePositions(segmentsAround, segmentsDown);
```

javascript

```
const normals = positions.slice();
```

javascript

```
const geometry = new THREE.BufferGeometry();
const positionNumComponents = 3;
const normalNumComponents = 3;

const positionAttribute = new THREE.BufferAttribute(positions, positionNumComponent
positionAttribute.setUsage(THREE.DynamicDrawUsage);
geometry.setAttribute(
    'position',
    positionAttribute);
geometry.setAttribute(
    'normal',
    new THREE.BufferAttribute(normals, normalNumComponents));
geometry.setIndex(indices);
```

javascript

```
const temp = new THREE.Vector3();

...

for (let i = 0; i < positions.length; i += 3) {
  const quad = (i / 12 | 0);
  const ringId = quad / segmentsAround | 0;
  const ringQuadId = quad % segmentsAround;
  const ringU = ringQuadId / segmentsAround;
  const angle = ringU * Math.PI * 2;
  temp.fromArray(normals, i);
  temp.multiplyScalar(THREE.MathUtils.lerp(1, 1.4, Math.sin(time + ringId + angle)
  temp.toArray(positions, i);
}
positionAttribute.needsUpdate = true;
```

Dynamic Attribute Setup Notes

I've highlighted a few differences. We save a reference to the position attribute. We also mark it as dynamic. This is a hint to THREE.js that we're going to be changing the contents of the attribute often.

In our render loop we update the positions based off their normals every frame.

And we set positionAttribute.needsUpdate to tell THREE.js to use our changes.

Conclusion

I hope these were useful examples of how to use BufferGeometry directly to make your own geometry and how to dynamically update the contents of a BufferAttribute.

Three.js and Shadertoy

GUIDE

Source: https://threejs.org/manual/en/shadertoy.html

A guide explaining how to integrate Shadertoy shaders into Three.js, covering shader setup, uniform handling, texture inputs, vertex shaders, and procedural textures.

Three.js and Shadertoy

Shadertoy is a famous website hosting amazing shader experiments. People often ask how they can use those shaders with Three.js.

It's important to recognize it's called Shader TOY for a reason. In general shadertoy shaders are not about best practices. Rather they are a fun challenge similar to say dwitter (write code in 140 characters) or js13kGames (make a game in 13k or less).

In the case of Shadertoy the puzzle is, write a function that for a given pixel location outputs a color that draws something interesting. It's a fun challenge and many of the result are amazing. But, it is not best practice.

Compare this amazing shadertoy shader that draws an entire city

Fullscreen on my GPU it runs at about 5 frames a second. Contrast that to a game like Cities: Skylines

This game runs 30-60 frames a second on the same machine because it uses more traditional techniques, drawing buildings made from triangles with textures on them, etc...

Still, let's go over using a Shadertoy shader with three.js.

This is the default shadertoy shader if you pick "New" on shadertoy.com, at least as of January 2019.

One thing important to understand about shaders is they are written in a language called GLSL (Graphics Library Shading Language) designed for 3D math which includes special types. Above we see vec4, vec2, vec3 as 3 such special types. A vec2 has 2

values, a vec3 3, a vec4 4 values. They can be addressed in a bunch of ways. The most common ways are with x, y, z, and w as in

Unlike JavaScript, GLSL is more like C/C++ where variables have to have their type declared so instead of var v = 1.2; it's float v = 1.2; declaring v to be a floating point number.

Explaining GLSL in detail is more than we can do in this article. For a quick overview see this article and maybe follow that up with this series.

It should be noted that, at least as of January 2019, shadertoy.com only concerns itself with fragment shaders. A fragment shader's responsibility is, given a pixel location output a color for that pixel.

Looking at the function above we can see the shader has an out parameter called fragColor. out stands for output. It's a parameter the function is expected to provide a value for. We need to set this to some color.

It also has an in (for input) parameter called fragCoord. This is the pixel coordinate that is about to be drawn. We can use that coordinate to decide on a color. If the canvas we're drawing to is 400x300 pixels then the function will be called 400x300 times or 120,000 times. Each time fragCoord will be a different pixel coordinate.

There are 2 more variables being used that are not defined in the code. One is iResolution. This is set to the resolution of the canvas. If the canvas is 400x300 then iResolution would be 400,300 so as the pixel coordinates change that makes uv go from 0.0 to 1.0 across and up the texture. Working with normalized values often makes things easier and so the majority of shadertoy shaders start with something like this.

The other undefined variable in the shader is iTime. This is the time since the page loaded in seconds.

In shader jargon these global variables are called uniform variables. They are called uniform because they don't change, they stay uniform from one iteration of the shader to the next. It's important to note all of them are specific to shadertoy. They not official GLSL variables. They are variables the makers of shadertoy made up.

The Shadertoy docs define several more. For now let's write something that handles the two being used in the shader above.

The first thing to do is let's make a single plane that fills the canvas. If you haven't read it yet we did this in the article on backgrounds so let's grab that example but remove the cubes. It's pretty short so here's the entire thing

As explained in the backgrounds article an OrthographicCamera with these parameters and a 2 unit plane will fill the canvas. For now all we'll get is a red canvas as our plane is using a red MeshBasicMaterial.

Now that we have something working let's add the shadertoy shader.

Above we declared the 2 uniform variables we talked about. Then we inserted the shader GLSL code from shadertoy. Finally we called mainImage passing it gl_FragColor and gl_FragCoord.xy. gl_FragColor is an official WebGL global variable the shader is responsible for setting to whatever color it wants the current pixel to be. gl_FragCoord is another official WebGL global variable that tells us the coordinate of the pixel we're currently choosing a color for.

We then need to setup three.js uniforms so we can supply values to the shader.

Each uniform in THREE.js has value parameter. That value has to match the type of the uniform.

Then we pass both the fragment shader and uniforms to a ShaderMaterial.

and before rendering we need to set the values of the uniforms

Note: I have no idea why iResolution is a vec3 and what's in the 3rd value is not documented on shadertoy.com. It's not used above so just setting it to 1 for now. ¯_(ツ)_/¯

This matches what we see on Shadertoy for a new shader, at least as of January 2019 ☺. What's the shader above doing?

- uv goes from 0 to 1.
- cos(uv.xyx) gives us 3 cosine values as a vec3. One for uv.x, another for uv.y and another for uv.x again.
- Adding in the time, cos(iTime+uv.xyx) makes them animate.
- Adding in vec3(0,2,4) as in cos(iTime+uv.xyx+vec3(0,2,4)) offsets the cosine waves
- cos goes from -1 to 1 so the 0.5 * 0.5 + cos(...) converts from -1 <-> 1 to 0.0 <-> 1.0

- the results are then used as the RGB color for the current pixel

A minor change will make it easier to see the cosine waves. Right now uv only goes from 0 to 1. A cosine repeats at 2π so let's make it go from 0 to 40 by multiplying by 40.0. That should make it repeat about 6.3 times.

Counting below I see about 6.3 repeats. We can see the blue between the red since it's offset by 4 via the +vec3(0,2,4). Without that the blue and red would overlap perfectly making purple.

Knowing how simple the inputs are and then seeing results like a city canal, a forest, a snail, a mushroom make the challenge all that much more impressive. Hopefully they also make it clear why it's not generally the right approach vs the more traditional ways of making scenes from triangles. The fact that so much math has to be put into computing the color of every pixel means those examples run very slow.

Some shadertoy shaders take textures as inputs like this one.

Passing a texture into a shader is similar to passing one into a normal material but we need to set up the texture on the uniforms.

First we'll add the uniform for the texture to the shader. They're referred to as sampler2D in GLSL.

Then we can load a texture like we covered here and assign the uniform's value.

So far we've been using Shadertoy shaders as they are used on Shadertoy.com, namely drawing to cover the canvas. There's no reason we need to limit it to just that use case though. The important part to remember is the functions people write on shadertoy generally just take a fragCoord input and a iResolution. fragCoord does not have to come from pixel coordinates, we could use something else like texture coordinates instead and could then use them kind of like other textures. This technique of using a function to generate textures is often called a procedural texture.

Let's change the shader above to do this. The simplest thing to do might be to take the texture coordinates that three.js normally supplies, multiply them by iResolution and pass that in for fragCoords.

To do that we add in a varying. A varying is a value passed from the vertex shader to the fragment shader that gets interpolated (or varied) between vertices. To use it in our fragment shader we declare it. Three.js refers to its texture coordinates as uv with the v in front meaning varying.

Then we need to also provide our own vertex shader. Here is a fairly common minimal three.js vertex shader. Three.js declares and will provide values for uv, projectionMatrix, modelViewMatrix, and position.

We need to pass the vertex shader to the ShaderMaterial

We can set the iResolution uniform value at init time since it will no longer change.

and we no longer need to set it at render time

Otherwise I copied back in the original camera and code that sets up 3 rotating cubes from the article on responsiveness. The result:

I hope this at least gets you started on how to use a shadertoy shader with three.js. Again, it's important to remember that most shadertoy shaders are an interesting challenge (draw everything with a single function) rather than the recommended way to actually display things in a performant way. Still, they are amazing, impressive, beautiful, and you can learn a ton by seeing how they work.

glsl

```glsl
// By iq: https://www.shadertoy.com/user/iq
// license: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported Lice
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Time varying pixel color
    vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

    // Output to screen
    fragColor = vec4(col,1.0);
}
```

glsl

```glsl
vec4 v1 = vec4(1.0, 2.0, 3.0, 4.0);
float v2 = v1.x + v1.y;  // adds 1.0 + 2.0
```

javascript

```
function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
  renderer.autoClearColor = false;

  const camera = new THREE.OrthographicCamera(
    -1, // left
     1, // right
     1, // top
    -1, // bottom
    -1, // near,
     1, // far,
  );
  const scene = new THREE.Scene();
  const plane = new THREE.PlaneGeometry(2, 2);
  const material = new THREE.MeshBasicMaterial({
      color: 'red',
  });
  scene.add(new THREE.Mesh(plane, material));

  function resizeRendererToDisplaySize(renderer) {
    const canvas = renderer.domElement;
    const width = canvas.clientWidth;
    const height = canvas.clientHeight;
    const needResize = canvas.width !== width || canvas.height !== height;
    if (needResize) {
      renderer.setSize(width, height, false);
    }
    return needResize;
  }

  function render() {
    resizeRendererToDisplaySize(renderer);

    renderer.render(scene, camera);

    requestAnimationFrame(render);
  }

  requestAnimationFrame(render);
}

main();
```

```
glsl
```

```
const fragmentShader =
#include <common>

uniform vec3 iResolution;
uniform float iTime;

// By iq: https://www.shadertoy.com/user/iq
// license: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported Lice
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    // Normalized pixel coordinates (from 0 to 1)
    vec2 uv = fragCoord/iResolution.xy;

    // Time varying pixel color
    vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));

    // Output to screen
    fragColor = vec4(col,1.0);
}

void main() {
  mainImage(gl_FragColor, gl_FragCoord.xy);
}
;
```

javascript

```
const uniforms = {
  iTime: { value: 0 },
  iResolution:  { value: new THREE.Vector3() },
};
```

javascript

```
-const material = new THREE.MeshBasicMaterial({
-    color: 'red',
-});
+const material = new THREE.ShaderMaterial({
+  fragmentShader,
+  uniforms,
+});
```

javascript

```
-function render() {
+function render(time) {
+  time *= 0.001;  // convert to seconds

  resizeRendererToDisplaySize(renderer);

+  const canvas = renderer.domElement;
+  uniforms.iResolution.value.set(canvas.width, canvas.height, 1);
+  uniforms.iTime.value = time;

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

glsl

```
-vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx+vec3(0,2,4));
+vec3 col = 0.5 + 0.5*cos(iTime+uv.xyx*40.0+vec3(0,2,4));
```

glsl

```
// By Daedelus: https://www.shadertoy.com/user/Daedelus
// license: Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported Lice
#define TIMESCALE 0.25
#define TILES 8
#define COLOR 0.7, 1.6, 2.8

void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec2 uv = fragCoord.xy / iResolution.xy;
    uv.x *= iResolution.x / iResolution.y;

    vec4 noise = texture2D(iChannel0, floor(uv * float(TILES)) / float(TILES));
    float p = 1.0 - mod(noise.r + noise.g + noise.b + iTime * float(TIMESCALE), 1.0
    p = min(max(p * 3.0 - 1.8, 0.1), 2.0);

    vec2 r = mod(uv * float(TILES), 1.0);
    r = vec2(pow(r.x - 0.5, 2.0), pow(r.y - 0.5, 2.0));
    p *= 1.0 - pow(min(1.0, 12.0 * dot(r, r)), 2.0);

    fragColor = vec4(COLOR, 1.0) * p;
}
```

glsl

```
const fragmentShader =
#include <common>

uniform vec3 iResolution;
uniform float iTime;
+uniform sampler2D iChannel0;

...
```

javascript

```
+const loader = new THREE.TextureLoader();
+const texture = loader.load('resources/images/bayer.png');
+texture.minFilter = THREE.NearestFilter;
+texture.magFilter = THREE.NearestFilter;
+texture.wrapS = THREE.RepeatWrapping;
+texture.wrapT = THREE.RepeatWrapping;
const uniforms = {
  iTime: { value: 0 },
  iResolution:  { value: new THREE.Vector3() },
+  iChannel0: { value: texture },
};
```

glsl

```
+varying vec2 vUv;

void main() {
-  mainImage(gl_FragColor, gl_FragCoord.xy);
+  mainImage(gl_FragColor, vUv * iResolution.xy);
}
```

glsl

```
const vertexShader =
  varying vec2 vUv;
  void main() {
    vUv = uv;
    gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );
  }
`;
```

javascript

```
const material = new THREE.ShaderMaterial({
  vertexShader,
  fragmentShader,
  uniforms,
});
```

javascript

```
const uniforms = {
  iTime: { value: 0 },
-  iResolution:  { value: new THREE.Vector3() },
+  iResolution:  { value: new THREE.Vector3(1, 1, 1) },
  iChannel0: { value: texture },
};
```

javascript

```
-const canvas = renderer.domElement;
-uniforms.iResolution.value.set(canvas.width, canvas.height, 1);
uniforms.iTime.value = time;
```

Physics, Games & Interaction

Picking

GUIDE

Source: https://threejs.org/manual/en/picking.html

This page explains two common methods for implementing picking (determining which object a user clicked) in THREE.js: CPU-based raycasting using the RayCaster class, and GPU-based picking using unique object colors rendered offscreen. It includes working code examples, discusses the tradeoffs of each approach, and demonstrates both implementations with a scene of 100 cubes.

Introduction to Picking

Picking refers to the process of figuring out which object a user clicked on or touched. There are tons of ways to implement picking each with their tradeoffs. Let's go over the 2 most common.

Probably the most common way of *picking* is by doing raycasting which means to *cast* a ray from the mouse through the frustum of the scene and computing which objects that ray intersects. Conceptually it's very simple.

First we'd take the position of the mouse. We'd convert that into world space by applying the camera's projection and orientation. We'd compute a ray from the near plane of the camera's frustum to the far plane. Then, for every triangle of every object in the scene we'd check if that ray intersects that triangle. If your scene has 1000 objects and each object has 1000 triangles then 1 million triangles will need to be checked.

A few optimizations would include first checking if the ray intersects with an object's bounding sphere or bounding box, the sphere or box that contains the entire object. If the ray doesn't intersect one of those then we don't have to check the triangles of that object.

THREE.js provides a `RayCaster` class that does exactly this.

Let's make a scene with a 100 objects and try picking them. We'll start with an example from the article on responsive pages.

Setting Up the Scene with a Camera Pole

We'll parent the camera to another object so we can spin that other object and the camera will move around the scene just like a selfie stick.

```
javascript
```

```
const fov = 60;
const aspect = 2;  // the canvas default
const near = 0.1;
const far = 200;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
camera.position.z = 30;

const scene = new THREE.Scene();
scene.background = new THREE.Color('white');

// put the camera on a pole (parent it to an object)
// so we can spin the pole to move the camera around the scene
const cameraPole = new THREE.Object3D();
scene.add(cameraPole);
cameraPole.add(camera);
```

Spinning the Camera Pole

In the `render` function we'll spin the camera pole.

javascript

```
cameraPole.rotation.y = time * .1;
```

Moving the Light with the Camera

Also let's put the light on the camera so the light moves with it.

javascript

```
scene.add(light);
camera.add(light);
```

Generating 100 Cubes

Let's generate 100 cubes with random colors in random positions, orientations, and scales.

javascript

```
const boxWidth = 1;
const boxHeight = 1;
const boxDepth = 1;
const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);

function rand(min, max) {
  if (max === undefined) {
    max = min;
    min = 0;
  }
  return min + (max - min) * Math.random();
}

function randomColor() {
  return `hsl(${rand(360) | 0}, ${rand(50, 100) | 0}%, 50%)`;
}

const numObjects = 100;
for (let i = 0; i < numObjects; ++i) {
  const material = new THREE.MeshPhongMaterial({
    color: randomColor(),
  });

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  cube.position.set(rand(-20, 20), rand(-20, 20), rand(-20, 20));
  cube.rotation.set(rand(Math.PI), rand(Math.PI), 0);
  cube.scale.set(rand(3, 6), rand(3, 6), rand(3, 6));
}
```

Creating the PickHelper Class

Let's make a simple class to manage the picking. You can see we create a `RayCaster` and then we can call the `pick` function to cast a ray through the scene. If the ray hits something we change the color of the first thing it hits.

```
javascript
```

```
class PickHelper {
  constructor() {
    this.raycaster = new THREE.Raycaster();
    this.pickedObject = null;
    this.pickedObjectSavedColor = 0;
  }
  pick(normalizedPosition, scene, camera, time) {
    // restore the color if there is a picked object
    if (this.pickedObject) {
      this.pickedObject.material.emissive.setHex(this.pickedObjectSavedColor);
      this.pickedObject = undefined;
    }

    // cast a ray through the frustum
    this.raycaster.setFromCamera(normalizedPosition, camera);
    // get the list of objects the ray intersected
    const intersectedObjects = this.raycaster.intersectObjects(scene.children);
    if (intersectedObjects.length) {
      // pick the first object. It's the closest one
      this.pickedObject = intersectedObjects[0].object;
      // save its color
      this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
      // set its emissive color to flashing red/yellow
      this.pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFFFF00 : 0x
    }
  }
}
```

Tracking Mouse Position

Of course we could call this function only when the user pressed the mouse *down* which is probably usually what you want but for this example we'll pick every frame whatever is under the mouse. To do this we first need to track where the mouse is.

Notice we're recording a normalized mouse position. Regardless of the size of the canvas we need a value that goes from -1 on the left to +1 on the right. Similarly we need a value that goes from -1 on the bottom to +1 on the top.

```
javascript
```

```
const pickPosition = {x: 0, y: 0};
clearPickPosition();

...

function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}

function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
  pickPosition.x = (pos.x / canvas.width ) *  2 - 1;
  pickPosition.y = (pos.y / canvas.height) * -2 + 1;  // note we flip Y
}

function clearPickPosition() {
  // unlike the mouse which always has a position
  // if the user stops touching the screen we want
  // to stop picking. For now we just pick a value
  // unlikely to pick something
  pickPosition.x = -100000;
  pickPosition.y = -100000;
}

window.addEventListener('mousemove', setPickPosition);
window.addEventListener('mouseout', clearPickPosition);
window.addEventListener('mouseleave', clearPickPosition);
```

Adding Mobile Touch Support

While we're at it lets support mobile as well.

javascript

```
window.addEventListener('touchstart', (event) => {
  // prevent the window from scrolling
  event.preventDefault();
  setPickPosition(event.touches[0]);
}, {passive: false});

window.addEventListener('touchmove', (event) => {
  setPickPosition(event.touches[0]);
});

window.addEventListener('touchend', clearPickPosition);
```

Calling PickHelper in the Render Function

And finally in our `render` function we call the `PickHelper`'s `pick` function.

```javascript
const pickHelper = new PickHelper();

function render(time) {
  time *= 0.001;  // convert to seconds;

  ...

  pickHelper.pick(pickPosition, scene, camera, time);

  renderer.render(scene, camera);

  ...
}
```

Issues with Raycasting

This appears to work great and it probably does for many use cases but there are several issues.

It's CPU based.

JavaScript is going through each object and checking if the ray intersects that object's bounding box or bounding sphere. If it does then JavaScript has to go through each and every triangle in that object and check if the ray intersects the triangle.

The good part of this is JavaScript can easily compute exactly where the ray intersected the triangle and provide us with that data. For example if you wanted to put a marker where the intersection happened.

The bad part is that's a lot of work for the CPU to do. If you have objects with lots of triangles it might be slow.

It doesn't handle any strange shaders or displacements.

If you have a shader that deforms or morphs the geometry JavaScript has no knowledge of that deformation and so will give the wrong answer. For example AFAIK you can't use this method with skinned objects.

It doesn't handle transparent holes.

As an example let's apply this texture to the cubes.

Demonstrating the Transparency Problem

We'll just make these changes.

```javascript
const loader = new THREE.TextureLoader();
const texture = loader.load('resources/images/frame.png');

const numObjects = 100;
for (let i = 0; i < numObjects; ++i) {
  const material = new THREE.MeshPhongMaterial({
    color: randomColor(),
    map: texture,
    transparent: true,
    side: THREE.DoubleSide,
    alphaTest: 0.1,
  });

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  ...
}
```

Why Transparent Picking Fails

Try to pick something through a box and you can't. This is because JavaScript can't easily look into the textures and materials and figure out if part of your object is really transparent or not.

Introducing GPU-Based Picking

A solution all of these issues is to use GPU based picking. Unfortunately while it is conceptually simple it is more complicated to use than the ray casting method above.

To do GPU picking we render each object in a unique color offscreen. We then look up the color of the pixel corresponding to the mouse position. The color tells us which object was picked.

This can solve issue 2 and 3 above. As for issue 1, speed, it really depends. Every object has to be drawn twice. Once to draw it for viewing and again to draw it for picking. It's possible with fancier solutions maybe both of those could be done at the same time but we're not going to try that.

One thing we can do though is since we're only going to be reading one pixel we can just setup the camera so only that one pixel is drawn. We can do this using `PerspectiveCamera.setViewOffset` which lets us tell THREE.js to compute a camera that just renders a smaller part of a larger rectangle. This should save some time.

To do this type of picking in THREE.js at the moment requires we create 2 scenes. One we will fill with our normal meshes. The other we'll fill with meshes that use our picking material.

Creating the Picking Scene

So, first create a second scene and make sure it clears to black.

```javascript
const scene = new THREE.Scene();
scene.background = new THREE.Color('white');
const pickingScene = new THREE.Scene();
pickingScene.background = new THREE.Color(0);
```

Creating Picking Cubes for Each Object

Then, for each cube we place in the main scene we make a corresponding "picking cube" at the same position as the original cube, put it in the `pickingScene`, and set its material to something that will draw the object's id as its color. Also we keep a map of ids to objects so when we look up an id later we can map it back to its corresponding object.

Note that we are abusing the `MeshPhongMaterial` here. By setting its `emissive` to our id and the `color` and `specular` to 0 that will end up rendering the id only where the texture's alpha is greater than `alphaTest`. We also need to set `blending` to `NoBlending` so that the id is not multiplied by alpha.

Note that abusing the `MeshPhongMaterial` might not be the best solution as it will still calculate all our lights when drawing the picking scene even though we don't need those calculations. A more optimized solution would make a custom shader that just writes the id where the texture's alpha is greater than `alphaTest`.

```javascript
```

```
const idToObject = {};
const numObjects = 100;
for (let i = 0; i < numObjects; ++i) {
  const id = i + 1;
  const material = new THREE.MeshPhongMaterial({
    color: randomColor(),
    map: texture,
    transparent: true,
    side: THREE.DoubleSide,
    alphaTest: 0.1,
  });

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);
  idToObject[id] = cube;

  cube.position.set(rand(-20, 20), rand(-20, 20), rand(-20, 20));
  cube.rotation.set(rand(Math.PI), rand(Math.PI), 0);
  cube.scale.set(rand(3, 6), rand(3, 6), rand(3, 6));

  const pickingMaterial = new THREE.MeshPhongMaterial({
    emissive: new THREE.Color().setHex(id, THREE.NoColorSpace),
    color: new THREE.Color(0, 0, 0),
    specular: new THREE.Color(0, 0, 0),
    map: texture,
    transparent: true,
    side: THREE.DoubleSide,
    alphaTest: 0.5,
    blending: THREE.NoBlending,
  });
  const pickingCube = new THREE.Mesh(geometry, pickingMaterial);
  pickingScene.add(pickingCube);
  pickingCube.position.copy(cube.position);
  pickingCube.rotation.copy(cube.rotation);
  pickingCube.scale.copy(cube.scale);
}
```

Updating Pick Position for Pixel-Based Picking

Because we're picking from pixels instead of ray casting we can change the code that sets the pick position to just use pixels.

javascript

```
function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
  pickPosition.x = pos.x;
  pickPosition.y = pos.y;
}
```

Creating the GPUPickHelper Class

Then let's change the `PickHelper` into a `GPUPickHelper`. It will use a `WebGLRenderTarget` like we covered the article on render targets. Our render target

here is only a single pixel in size, 1x1.

```javascript
```

```
class GPUPickHelper {
  constructor() {
    // create a 1x1 pixel render target
    this.pickingTexture = new THREE.WebGLRenderTarget(1, 1);
    this.pixelBuffer = new Uint8Array(4);
    this.pickedObject = null;
    this.pickedObjectSavedColor = 0;
  }
  pick(cssPosition, scene, camera, time) {
    const {pickingTexture, pixelBuffer} = this;

    // restore the color if there is a picked object
    if (this.pickedObject) {
      this.pickedObject.material.emissive.setHex(this.pickedObjectSavedColor);
      this.pickedObject = undefined;
    }

    // set the view offset to represent just a single pixel under the mouse
    const pixelRatio = renderer.getPixelRatio();
    camera.setViewOffset(
        renderer.getContext().drawingBufferWidth,   // full width
        renderer.getContext().drawingBufferHeight,  // full top
        cssPosition.x * pixelRatio | 0,             // rect x
        cssPosition.y * pixelRatio | 0,             // rect y
        1,                                          // rect width
        1,                                          // rect height
    );
    // render the scene
    renderer.setRenderTarget(pickingTexture)
    renderer.render(scene, camera);
    renderer.setRenderTarget(null);

    // clear the view offset so rendering returns to normal
    camera.clearViewOffset();
    //read the pixel
    renderer.readRenderTargetPixels(
        pickingTexture,
        0,   // x
        0,   // y
        1,   // width
        1,   // height
        pixelBuffer);

    const id =
        (pixelBuffer[0] << 16) |
        (pixelBuffer[1] <<  8) |
        (pixelBuffer[2]      );

    const intersectedObject = idToObject[id];
    if (intersectedObject) {
      // pick the first object. It's the closest one
      this.pickedObject = intersectedObject;
      // save its color
      this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
      // set its emissive color to flashing red/yellow
      this.pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFFFF00 : 0x
    }
  }
}
```

Using the GPUPickHelper

Then we just need to use it, and pass it the `pickScene` instead of the `scene`. And now it should let you pick through the transparent parts.

```javascript
const pickHelper = new GPUPickHelper();

  pickHelper.pick(pickPosition, pickScene, camera, time);
```

Conclusion

I hope that gives some idea of how to implement picking. In a future article maybe we can cover how to manipulate objects with the mouse.

Indexed Textures for Picking and Color

GUIDE

Source: https://threejs.org/manual/en/indexed-textures.html

A tutorial on using indexed textures in Three.js for GPU-based picking and per-country highlighting on a 3D globe, including shader modifications via onBeforeCompile to implement paletted graphics for highlighting selections.

Introduction

This article is a continuation of an article about aligning html elements to 3d. If you haven't read that yet you should start there before continuing here.

Sometimes using three.js requires coming up with creative solutions. I'm not sure this is a great solution but I thought I'd share it and you can see if it suggests any solutions for your needs.

In the previous article we displayed country names around a 3d globe. How would we go about letting the user select a country and show their selection?

The Problem: Picking Countries on a Globe

The first idea that comes to mind is to generate geometry for each country. We could use a picking solution like we covered before. We'd build 3D geometry for each country. If the user clicks on the mesh for that country we'd know what country was clicked.

So, just to check that solution I tried generating 3D meshes of all the countries using the same data I used to generate the outlines in the previous article. The result was a 15.5meg binary GLTF (.glb) file. Making the user download 15.5meg sounds like too much to me.

There are lots of ways to compress the data. The first would probably be to apply some algorithm to lower the resolution of the outlines. I didn't spend any time pursuing that solution. For borders of the USA that's probably a huge win. For a borders of Canada probably much less.

Another solution would be to use just actual data compression. For example gzipping the file brought it down to 11meg. That's 30% less but arguably not enough.

We could store all the data as 16bit ranged values instead of 32bit float values. Or we could use something like draco compression and maybe that would be enough. I didn't check and I would encourage you to check yourself and tell me how it goes as I'd love to know. 😄

The Solution: Index Texture for Picking

In my case I thought about the GPU picking solution we covered at the end of the article on picking. In that solution we drew every mesh with a unique color that represented that mesh's id. We then drew all the meshes and looked at the color that was clicked on.

Taking inspiration from that we could pre-generate a map of countries where each country's color is its index number in our array of countries. We could then use a similar GPU picking technique. We'd draw the globe off screen using this index texture. Looking at the color of the pixel the user clicks would tell us the country id.

So, I wrote some code to generate such a texture. Here it is.

Note: The data used to generate this texture comes from this website and is therefore licensed as CC-BY-SA.

It's only 217k, much better than the 14meg for the country meshes. In fact we could probably even lower the resolution but 217k seems good enough for now.

So let's try using it for picking countries.

Creating a Picking Scene

Grabbing code from the gpu picking example we need a scene for picking.

```javascript
const pickingScene = new THREE.Scene();
pickingScene.background = new THREE.Color(0);
```

Adding the Globe with Index Texture to the Picking Scene

and we need to add the globe with the our index texture to the picking scene.

```javascript
{
  const loader = new THREE.TextureLoader();
  const geometry = new THREE.SphereGeometry(1, 64, 32);

+  const indexTexture = loader.load('resources/data/world/country-index-texture.png
+  indexTexture.minFilter = THREE.NearestFilter;
+  indexTexture.magFilter = THREE.NearestFilter;
+
+  const pickingMaterial = new THREE.MeshBasicMaterial({map: indexTexture});
+  pickingScene.add(new THREE.Mesh(geometry, pickingMaterial));

  const texture = loader.load('resources/data/world/country-outlines-4k.png', rende
  const material = new THREE.MeshBasicMaterial({map: texture});
  scene.add(new THREE.Mesh(geometry, material));
}
```

The GPUPickHelper Class

Then let's copy over the `GPUPickingHelper` class we used before with a few minor changes.

```javascript
```

```
class GPUPickHelper {
  constructor() {
    // create a 1x1 pixel render target
    this.pickingTexture = new THREE.WebGLRenderTarget(1, 1);
    this.pixelBuffer = new Uint8Array(4);
-    this.pickedObject = null;
-    this.pickedObjectSavedColor = 0;
  }
  pick(cssPosition, scene, camera) {
    const {pickingTexture, pixelBuffer} = this;

    // set the view offset to represent just a single pixel under the mouse
    const pixelRatio = renderer.getPixelRatio();
    camera.setViewOffset(
        renderer.getContext().drawingBufferWidth,   // full width
        renderer.getContext().drawingBufferHeight,  // full top
        cssPosition.x * pixelRatio | 0,             // rect x
        cssPosition.y * pixelRatio | 0,             // rect y
        1,                                          // rect width
        1,                                          // rect height
    );
    // render the scene
    renderer.setRenderTarget(pickingTexture);
    renderer.render(scene, camera);
    renderer.setRenderTarget(null);
    // clear the view offset so rendering returns to normal
    camera.clearViewOffset();
    //read the pixel
    renderer.readRenderTargetPixels(
        pickingTexture,
        0,   // x
        0,   // y
        1,   // width
        1,   // height
        pixelBuffer);

+    const id =
+        (pixelBuffer[0] << 16) |
+        (pixelBuffer[1] <<  8) |
+        (pixelBuffer[2] <<  0);
+
+    return id;
-    const id =
-        (pixelBuffer[0] << 16) |
-        (pixelBuffer[1] <<  8) |
-        (pixelBuffer[2]      );
-    const intersectedObject = idToObject[id];
-    if (intersectedObject) {
-      // pick the first object. It's the closest one
-      this.pickedObject = intersectedObject;
-      // save its color
-      this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
-      // set its emissive color to flashing red/yellow
-      this.pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFFFF00 : 0
-    }
  }
}
```

Picking Countries

Now we can use that to pick countries.

The code above sets/unsets the `selected` property on the array of countries. If `shift` or `ctrl` or `cmd` is pressed then you can select more than one country.

javascript

```
const pickHelper = new GPUPickHelper();

function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}

function pickCountry(event) {
  // exit if we have not loaded the data yet
  if (!countryInfos) {
    return;
  }

  const position = getCanvasRelativePosition(event);
  const id = pickHelper.pick(position, pickingScene, camera);
  if (id > 0) {
    // we clicked a country. Toggle its 'selected' property
    const countryInfo = countryInfos[id - 1];
    const selected = !countryInfo.selected;
    // if we're selecting this country and modifiers are not
    // pressed unselect everything else.
    if (selected && !event.shiftKey && !event.ctrlKey && !event.metaKey) {
      unselectAllCountries();
    }
    numCountriesSelected += selected ? 1 : -1;
    countryInfo.selected = selected;
  } else if (numCountriesSelected) {
    // the ocean or sky was clicked
    unselectAllCountries();
  }
  requestRenderIfNotRequested();
}

function unselectAllCountries() {
  numCountriesSelected = 0;
  countryInfos.forEach((countryInfo) => {
    countryInfo.selected = false;
  });
}

canvas.addEventListener('pointerup', pickCountry);
```

Showing Selected Countries

All that's left is showing the selected countries. For now let's just update the labels.

javascript

```
function updateLabels() {
  // exit if we have not loaded the data yet
  if (!countryInfos) {
    return;
  }

  const large = settings.minArea * settings.minArea;
  // get a matrix that represents a relative orientation of the camera
  normalMatrix.getNormalMatrix(camera.matrixWorldInverse);
  // get the camera's position
  camera.getWorldPosition(cameraPosition);
  for (const countryInfo of countryInfos) {
-    const {position, elem, area} = countryInfo;
-    // large enough?
-    if (area < large) {
+    const {position, elem, area, selected} = countryInfo;
+    const largeEnough = area >= large;
+    const show = selected || (numCountriesSelected === 0 && largeEnough);
+    if (!show) {
      elem.style.display = 'none';
      continue;
    }

    ...
```

Paletted Graphics / Indexed Color

The code stills shows countries based on their area but if you click one just that one will have a label.

So that seems like a reasonable solution for picking countries but what about highlighting the selected countries?

For that we can take inspiration from paletted graphics.

Paletted graphics or Indexed Color is what older systems like the Atari 800, Amiga, NES, Super Nintendo, and even older IBM PCs used. Instead of storing bitmaps as RGBA colors 8bits per color, 32 bytes per pixel or more, they stored bitmaps as 8bit values or less. The value for each pixel was an index into a palette. So for example a value of 3 in the image means "display color 3". What color color#3 is is defined somewhere else called a "palette".

In JavaScript you can think of it like this

javascript

```
const face7x7PixelImageData = [
  0, 1, 1, 1, 1, 1, 0,
  1, 0, 0, 0, 0, 0, 1,
  1, 0, 2, 0, 2, 0, 1,
  1, 0, 0, 0, 0, 0, 1,
  1, 0, 3, 3, 3, 0, 1,
  1, 0, 0, 0, 0, 0, 1,
  0, 1, 1, 1, 1, 1, 1,
];

const palette = [
  [255, 255, 255],  // white
  [  0,   0,   0],  // black
  [  0, 255, 255],  // cyan
  [255,   0,   0],  // red
];
```

Applying Palettes to Country Textures

Where each pixel in the image data is an index into palette. If you interpreted the image data through the palette above you'd get this image

In our case we already have a texture above that has a different id per country. So, we could use that same texture through a palette texture to give each country its own color. By changing the palette texture we can color each individual country. For example by setting the entire palette texture to black and then for one country's entry in the palette a different color, we can highlight just that country.

To do paletted index graphics requires some custom shader code. Let's modify the default shaders in three.js. That way we can use lighting and other features if we want.

Like we covered in the article on animating lots of objects we can modify the default shaders by adding a function to a material's `onBeforeCompile` property.

Default Fragment Shader

The default fragment shader looks something like this before compiling.

```
glsl
```

```
#include <common>
#include <color_pars_fragment>
#include <uv_pars_fragment>
#include <map_pars_fragment>
#include <alphamap_pars_fragment>
#include <aomap_pars_fragment>
#include <lightmap_pars_fragment>
#include <envmap_pars_fragment>
#include <fog_pars_fragment>
#include <specularmap_pars_fragment>
#include <logdepthbuf_pars_fragment>
#include <clipping_planes_pars_fragment>
void main() {
    #include <clipping_planes_fragment>
    vec4 diffuseColor = vec4( diffuse, opacity );
    #include <logdepthbuf_fragment>
    #include <map_fragment>
    #include <color_fragment>
    #include <alphamap_fragment>
    #include <alphatest_fragment>
    #include <specularmap_fragment>
    ReflectedLight reflectedLight = ReflectedLight( vec3( 0.0 ), vec3( 0.0 ), vec3(
    #ifdef USE_LIGHTMAP
        reflectedLight.indirectDiffuse += texture2D( lightMap, vLightMapUv ).xyz *
    #else
        reflectedLight.indirectDiffuse += vec3( 1.0 );
    #endif
    #include <aomap_fragment>
    reflectedLight.indirectDiffuse *= diffuseColor.rgb;
    vec3 outgoingLight = reflectedLight.indirectDiffuse;
    #include <envmap_fragment>
    gl_FragColor = vec4( outgoingLight, diffuseColor.a );
    #include <premultiplied_alpha_fragment>
    #include <tonemapping_fragment>
    #include <colorspace_fragment>
    #include <fog_fragment>
}
```

Modifying the Shader with onBeforeCompile

Digging through all those snippets we find that three.js uses a variable called `diffuseColor` to manage the base material color. It sets this in the `<color_fragment>` snippet so we should be able to modify it after that point.

`diffuseColor` at that point in the shader should already be the color from our outline texture so we can look up the color from a palette texture and mix them for the final result.

Like we did before we'll make an array of search and replacement strings and apply them to the shader in `Material.onBeforeCompile`.

Above can see above we add 3 uniforms, `indexTexture`, `paletteTexture`, and `paletteTextureWidth`. We get a color from the `indexTexture` and convert it to an

index. `vUv` is the texture coordinates provided by three.js. We then use that index to get a color out of the palette texture. We then mix the result with the current `diffuseColor`. The `diffuseColor` at this point is our black and white outline texture so if we add the 2 colors we'll get white outlines. If we subtract the current diffuse color we'll get black outlines.

javascript

```
{
  const loader = new THREE.TextureLoader();
  const geometry = new THREE.SphereGeometry(1, 64, 32);

  const indexTexture = loader.load('resources/data/world/country-index-texture.png'
  indexTexture.minFilter = THREE.NearestFilter;
  indexTexture.magFilter = THREE.NearestFilter;

  const pickingMaterial = new THREE.MeshBasicMaterial({map: indexTexture});
  pickingScene.add(new THREE.Mesh(geometry, pickingMaterial));

+  const fragmentShaderReplacements = [
+    {
+      from: '#include <common>',
+      to:
+        #include <common>
+        uniform sampler2D indexTexture;
+        uniform sampler2D paletteTexture;
+        uniform float paletteTextureWidth;
+      ,
+    },
+    {
+      from: '#include <color_fragment>',
+      to:
+        #include <color_fragment>
+        {
+          vec4 indexColor = texture2D(indexTexture, vUv);
+          float index = indexColor.r * 255.0 + indexColor.g * 255.0 * 256.0;
+          vec2 paletteUV = vec2((index + 0.5) / paletteTextureWidth, 0.5);
+          vec4 paletteColor = texture2D(paletteTexture, paletteUV);
+          // diffuseColor.rgb += paletteColor.rgb;   // white outlines
+          diffuseColor.rgb = paletteColor.rgb - diffuseColor.rgb;  // black outlin
+        }
+      ,
+    },
+  ];

  const texture = loader.load('resources/data/world/country-outlines-4k.png', rende
  const material = new THREE.MeshBasicMaterial({map: texture});
+  material.onBeforeCompile = function(shader) {
+    fragmentShaderReplacements.forEach((rep) => {
+      shader.fragmentShader = shader.fragmentShader.replace(rep.from, rep.to);
+    });
+  };
  scene.add(new THREE.Mesh(geometry, material));
}
```

Setting Up the Palette Texture

Before we can render we need to setup the palette texture and these 3 uniforms.

For the palette texture it just needs to be wide enough to hold one color per country + one for the ocean (id = 0). There are 240 something countries. We could wait until the list of countries loads to get an exact number or look it up. There's not much harm in just picking some larger number so let's choose 512.

Here's the code to create the palette texture

javascript

```
const maxNumCountries = 512;
const paletteTextureWidth = maxNumCountries;
const paletteTextureHeight = 1;
const palette = new Uint8Array(paletteTextureWidth * 4);
const paletteTexture = new THREE.DataTexture(
    palette, paletteTextureWidth, paletteTextureHeight);
paletteTexture.minFilter = THREE.NearestFilter;
paletteTexture.magFilter = THREE.NearestFilter;
```

DataTexture Description

A `DataTexture` let's us give a texture raw data. In this case we're giving it 512 RGBA colors, 4 bytes each where each byte is red, green, and blue respectively using values that go from 0 to 255.

Let's fill it with random colors just to see it work

javascript

```
for (let i = 1; i < palette.length; ++i) {
  palette[i] = Math.random() * 256;
}
// set the ocean color (index #0)
palette.set([100, 200, 255, 255], 0);
paletteTexture.needsUpdate = true;
```

Updating the Palette Texture

Anytime we want three.js to update the palette texture with the contents of the `palette` array we need to set `paletteTexture.needsUpdate` to `true`.

And then we still need to set the uniforms on the material.

javascript

```
const geometry = new THREE.SphereGeometry(1, 64, 32);
const material = new THREE.MeshBasicMaterial({map: texture});
material.onBeforeCompile = function(shader) {
  fragmentShaderReplacements.forEach((rep) => {
    shader.fragmentShader = shader.fragmentShader.replace(rep.from, rep.to);
  });
+  shader.uniforms.paletteTexture = {value: paletteTexture};
+  shader.uniforms.indexTexture = {value: indexTexture};
+  shader.uniforms.paletteTextureWidth = {value: paletteTextureWidth};
};
scene.add(new THREE.Mesh(geometry, material));
```

Color Helper Function

Now that we can see the index and palette textures are working let's manipulate the palette for highlighting.

First let's make function that will let us pass in a three.js style color and give us values we can put in the palette texture.

javascript

```
const tempColor = new THREE.Color();
function get255BasedColor(color) {
  tempColor.set(color);
  const base = tempColor.toArray().map(v => v * 255);
  base.push(255); // alpha
  return base;
}
```

Building the Palette

Calling it like this `color = get255BasedColor('red')` will return an array like `[255, 0, 0, 255]`.

Next let's use it to make a few colors and fill out the palette.

javascript

```
const selectedColor = get255BasedColor('red');
const unselectedColor = get255BasedColor('#444');
const oceanColor = get255BasedColor('rgb(100,200,255)');
resetPalette();

function setPaletteColor(index, color) {
  palette.set(color, index * 4);
}

function resetPalette() {
  // make all colors the unselected color
  for (let i = 1; i < maxNumCountries; ++i) {
    setPaletteColor(i, unselectedColor);
  }

  // set the ocean color (index #0)
  setPaletteColor(0, oceanColor);
  paletteTexture.needsUpdate = true;
}
```

Updating Palette on Country Selection

Now let's use those functions to update the palette when a country is selected

```
javascript
```

```
function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}

function pickCountry(event) {
  // exit if we have not loaded the data yet
  if (!countryInfos) {
    return;
  }

  const position = getCanvasRelativePosition(event);
  const id = pickHelper.pick(position, pickingScene, camera);
  if (id > 0) {
    const countryInfo = countryInfos[id - 1];
    const selected = !countryInfo.selected;
    if (selected && !event.shiftKey && !event.ctrlKey && !event.metaKey) {
      unselectAllCountries();
    }
    numCountriesSelected += selected ? 1 : -1;
    countryInfo.selected = selected;
+    setPaletteColor(id, selected ? selectedColor : unselectedColor);
+    paletteTexture.needsUpdate = true;
  } else if (numCountriesSelected) {
    unselectAllCountries();
  }
  requestRenderIfNotRequested();
}

function unselectAllCountries() {
  numCountriesSelected = 0;
  countryInfos.forEach((countryInfo) => {
    countryInfo.selected = false;
  });
+  resetPalette();
}
```

Handling Click vs Drag

One minor thing is we can't spin the globe without changing the selection state. If we select a country and then want to rotate the globe the selection will change.

Let's try to fix that. Off the top of my head we can check 2 things. How much time passed between clicking and letting go. Another is did the user actually move the mouse. If the time is short or if they didn't move the mouse then it was probably a click. Otherwise they were probably trying to drag the globe.

```
javascript
```

```
+const maxClickTimeMs = 200;
+const maxMoveDeltaSq = 5 * 5;
+const startPosition = {};
+let startTimeMs;
+
+function recordStartTimeAndPosition(event) {
+  startTimeMs = performance.now();
+  const pos = getCanvasRelativePosition(event);
+  startPosition.x = pos.x;
+  startPosition.y = pos.y;
+}

function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}

function pickCountry(event) {
  // exit if we have not loaded the data yet
  if (!countryInfos) {
    return;
  }

+  // if it's been a moment since the user started
+  // then assume it was a drag action, not a select action
+  const clickTimeMs = performance.now() - startTimeMs;
+  if (clickTimeMs > maxClickTimeMs) {
+    return;
+  }
+
+  // if they moved assume it was a drag action
+  const position = getCanvasRelativePosition(event);
+  const moveDeltaSq = (startPosition.x - position.x) ** 2 +
+                      (startPosition.y - position.y) ** 2;
+  if (moveDeltaSq > maxMoveDeltaSq) {
+    return;
+  }

-  const position = {x: event.clientX, y: event.clientY};
  const id = pickHelper.pick(position, pickingScene, camera);
  if (id > 0) {
    const countryInfo = countryInfos[id - 1];
    const selected = !countryInfo.selected;
    if (selected && !event.shiftKey && !event.ctrlKey && !event.metaKey) {
      unselectAllCountries();
    }
    numCountriesSelected += selected ? 1 : -1;
    countryInfo.selected = selected;
    setPaletteColor(id, selected ? selectedColor : unselectedColor);
    paletteTexture.needsUpdate = true;
  } else if (numCountriesSelected) {
    unselectAllCountries();
  }
  requestRenderIfNotRequested();
}

function unselectAllCountries() {
  numCountriesSelected = 0;
```

```
  countryInfos.forEach((countryInfo) => {
    countryInfo.selected = false;
  });
  resetPalette();
}

+canvas.addEventListener('pointerdown', recordStartTimeAndPosition);
canvas.addEventListener('pointerup', pickCountry);
```

Conclusion

I'm not a UX expert so I'd love to hear if there is a better solution.

I hope that gave you some idea of how indexed graphics can be useful and how you can modify the shaders three.js makes to add simple features. How to use GLSL, the language the shaders are written in, is too much for this article. There are a few links to some info in the article on post processing.

Physics

GUIDE

Source: https://threejs.org/manual/en/physics.html

An overview of how to integrate physics engines into three.js projects, covering three main approaches: using the official physics addons, third-party JS/TS libraries, and WASM-based engines.

Physics

Physics engines allow you to simulate physical phenomena like gravity, collisions, and forces within your 3D capabilities. In a typical three.js scene, objects are moved by directly modifying their position or rotation. When using a physics engine, however, you create a parallel physics world where bodies react to forces and collisions. You then synchronize the three.js meshes with these physics bodies on every frame, creating the illusion of a physically simulated environment.

It should be noted that the physics engine does not necessarily have to be updated every frame. Usually, to keep experiences consistent, physics are updated at fixed time steps. For instance, it could be that we are running the game loop at 60fps but the physics engine at 30fps (that is 1/30=3.33ms) while updating the three.js meshes' transforms (e.g., positions and rotations) with the most recent state from the physics engine.

Physics simulations are particularly useful for games, interactive visualizations, and any application requiring realistic object behavior, such as objects falling, bouncing, or sliding.

Integration Approaches

There are three main ways to integrate a physics engine into a three.js project:

1. Using Three.js Physics Addons

Three.js provides wrapper classes for several popular physics engines in the *examples/jsm/physics* directory. These addons simplify the setup process by handling the initialization of the physics world and the synchronization of meshes.

Available addons include:

- **AmmoPhysics:** A wrapper for Ammo.js (Bullet Physics).
- **JoltPhysics:** A wrapper for Jolt Physics.
- **RapierPhysics:** A wrapper for Rapier.

These addons effectively hide much of the complexity of the underlying engines. For standard use cases, they offer a very quick way to get started.

Examples

- physics / ammo / instancing
- physics / jolt / instancing
- physics / rapier / instancing

2. Using 3rd-Party Physics JS/TS Libraries

Many physics engines are written directly in JavaScript or TypeScript and are designed to work easily with the web ecosystem. Libraries like **cannon-es** are popular choices because they are lightweight and easy to integrate specifically with three.js.

When using these libraries, you instantiate the physics world and bodies yourself, then manually copy the position and quaternion from the physics body to the three.js mesh in your animation loop.

Projects

- cannon-es: A lightweight 3D physics engine purely written in JS/TS. Under an MIT license. Apparently no longer maintained (latest commit more than a couple years ago).
- cannon.js: A lightweight 3D physics engine purely written in JavaScript. Under an MIT license. No longer maintained (latest commit more than a couple years ago). Consider using its more recent fork cannon-es.
- phy: Physics engine for three.js purely written in JavaScript. Under an MIT license. Latest commit. Currently maintained.
- Oimo.js: A no-longer maintained lightweight 3D physics engine written purely in JavaScript. Under an MIT license. Consider using phy instead (as per the author's advise).

It should also be noted that there are a couple of 3D physics engines that are seemingly written in JS/TS but are in reality calling other standalone 3D physics engines. For example:

- Physijs: Calls ammo.js under the hood to do physics work in a separate thread (using web workers). Under MIT license. Apparently no longer maintained (latest commit more than a couple of years ago).
- enable3d: 3D physics framework for three.js built on top of ammo.js. Under LGPL-3.0 license. Apparently maintained.

3. Importing WASM-based Engines

For maximum performance, stability, and precision, especially with complex simulations, you can use physics engines written in C++ or Rust (or any other language that supports WASM) that have been compiled to WebAssembly (WASM). Engines like **Ammo.js** (a port of Bullet Physics) and **Rapier** fall into this category.

While this approach offers the most features and best performance, it often requires more setup code to handle the WASM memory management and interaction with the physics API directly.

Examples

- physics / ammo / break
- physics / ammo / cloth
- physics / ammo / rope
- physics / ammo / terrain

- physics / ammo / volume

Projects

- JoltPhysics: A multi core friendly rigid body physics and collision detection library. Written in C++. Under MIT license. Actively maintained. Proven with its usage in world-renowned titles and game engines including: Horizon Forbidden West, Death Stranding 2, and official supported by the Godot game engine.
- PhysX: Industry-standard realtime 3D physics engine provided by NVIDIA. Under BSD-3-Clause license. Actively maintained and very stable.
- Rapier: 2D and 3D physics engines focused on performance. Written in Rust. Under an MIT license. Actively maintained.
- Bullet: Real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc. Written in C++. Under a ZLIB license. Potentially no longer maintained.

Some of these multi-platform 3D physics engines have a ready-to-use WASM port, including:

- JoltPhysics.js: Port of JoltPhysics to JavaScript using Emscripten. Under MIT license. Currently maintained.
- physx-js-webidl: Javascript WASM bindings for Nvidia PhysX. Under MIT license. Currently maintained.
- Rapier.js: Official JavaScript bindings for the Rapier physics engine. Under Apache-2.0 license. Actively maintained.
- Ammo.js: Direct port of the Bullet physics engine to JavaScript using Emscripten. No longer maintained (latest commit a couple of years ago). Under an MIT-like custom permissive license.

Making a Game

GUIDE

Source: https://threejs.org/manual/en/game.html

A comprehensive guide on building a game with three.js, covering model loading with LoadingManager, animation playback, Entity Component System architecture, input systems, camera frustum checks, and a Finite State Machine for AI behavior.

Introduction

Many people want to write games using three.js. This article will hopefully give you some ideas on how to start.

At least at the time I'm writing this article it's probably going to be the longest article on this site. It's possible the code here is massively over engineered but as I wrote each new feature I'd run into a problem that needed a solution I'm used to from other games I've written. In other words each new solution seemed important so I'll try to show why. Of course the smaller your game the less you might need some of the solutions shown here but this is a pretty small game and yet with the complexities of 3D characters many things take more organization than they might with 2D characters.

As an example if you're making PacMan in 2D, when PacMan turns a corner that happens instantly at 90 degrees. There is no in-between step. But in a 3D game often we need the character to rotate over several frames. That simple change can add a bunch of complexity and require different solutions.

The majority of the code here will not really be three.js and that's important to note, three.js is not a game engine. Three.js is a 3D library. It provides a scene graph and features for displaying 3D objects added to that scene graph but it does not provide all the other things needed to make a game. No collisions, no physics, no input systems, no path finding, etc, etc... So, we'll have to provide those things ourselves.

I ended up writing quite a bit of code to make this simple unfinished game like thing and again, it's certainly possible I over engineered and there are simpler solutions but I feel like I actually didn't write enough code and hopefully I can explain what I think is missing.

Many of the ideas here are heavily influenced by Unity. If you're not familiar with Unity that probably does not matter. I only bring it up as 10s of 1000s of games have shipped using these ideas.

Let's start with the three.js parts. We need to load models for our game.

At opengameart.org I found this animated knight model by quaternius

quaternius also made these animated animals.

These seem like good models to start with so the first thing we need to do is load them.

Loading Models with LoadingManager

We covered loading glTF files before. The difference this time is we need to load multiple models and we can't start the game until all the models are loaded.

Fortunately three.js provides the LoadingManager just for this purpose. We create a LoadingManager and pass it to the other loaders. The LoadingManager provides both onProgress and onLoad properties we can attach callbacks to. The onLoad callback will be called when all files have been loaded. The onProgress callback as called after each individual file arrives to give as a chance to show loading progress.

Starting with the code from loading a glTF file I removed all the code related to framing the scene and added this code to load all models.

This code will load all the models above and the LoadingManager will call init when done. We'll use the models object later to let us access the loaded models so the GLTFLoader callback for each individual model attaches the loaded data to that model's info.

javascript

```
const manager = new THREE.LoadingManager();
manager.onLoad = init;
const models = {
  pig:    { url: 'resources/models/animals/Pig.gltf' },
  cow:    { url: 'resources/models/animals/Cow.gltf' },
  llama:  { url: 'resources/models/animals/Llama.gltf' },
  pug:    { url: 'resources/models/animals/Pug.gltf' },
  sheep:  { url: 'resources/models/animals/Sheep.gltf' },
  zebra:  { url: 'resources/models/animals/Zebra.gltf' },
  horse:  { url: 'resources/models/animals/Horse.gltf' },
  knight: { url: 'resources/models/knight/KnightCharacter.gltf' },
};
{
  const gltfLoader = new GLTFLoader(manager);
  for (const model of Object.values(models)) {
    gltfLoader.load(model.url, (gltf) => {
      model.gltf = gltf;
    });
  }
}

function init() {
  // TBD
}
```

Adding a Progress Bar

All the models with all their animation are currently about 6.6meg. That's a pretty big download. Assuming your server supports compression (the server this site runs on does) it's able to compress them to around 1.4meg. That's definitely better than

6.6meg bit it's still not a tiny amount of data. It would probably be good if we added a progress bar so the user has some idea how much longer they have to wait.

So, let's add an onProgress callback. It will be called with 3 arguments, the url of the last loaded object and then the number of items loaded so far as well as the total number of items.

Let's setup some HTML for a loading bar

We'll look up the #progressbar div and we can set the width from 0% to 100% to show our progress. All we need to do is set that in our callback.

We already setup init to be called when all the models are loaded so we can turn off the progress bar by hiding the #loading element.

Here's a bunch of CSS for styling the bar. The CSS makes the #loading div the full size of the page and centers its children. The CSS makes a .progress area to contain the progress bar. The CSS also gives the progress bar a CSS animation of diagonal stripes.

```html
 <body>
   <canvas id="c"></canvas>
+  <div id="loading">
+    <div>
+      <div>...loading...</div>
+      <div class="progress"><div id="progressbar"></div></div>
+    </div>
+  </div>
</body>
```

```javascript
const manager = new THREE.LoadingManager();
manager.onLoad = init;

+const progressbarElem = document.querySelector('#progressbar');
+manager.onProgress = (url, itemsLoaded, itemsTotal) => {
+  progressbarElem.style.width = ${itemsLoaded / itemsTotal * 100 | 0}%;
+};
```

```javascript
```

```
function init() {
+  // hide the loading bar
+  const loadingElem = document.querySelector('#loading');
+  loadingElem.style.display = 'none';
}
```

css

```css
#loading {
  position: absolute;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  display: flex;
  align-items: center;
  justify-content: center;
  text-align: center;
  font-size: xx-large;
  font-family: sans-serif;
}
#loading>div>div {
  padding: 2px;
}
.progress {
  width: 50vw;
  border: 1px solid black;
}
#progressbar {
  width: 0;
  transition: width ease-out .5s;
  height: 1em;
  background-color: #888;
  background-image: linear-gradient(
    -45deg,
    rgba(255, 255, 255, .5) 25%,
    transparent 25%,
    transparent 50%,
    rgba(255, 255, 255, .5) 50%,
    rgba(255, 255, 255, .5) 75%,
    transparent 75%,
    transparent
  );
  background-size: 50px 50px;
  animation: progressanim 2s linear infinite;
}

@keyframes progressanim {
  0% {
    background-position: 50px 50px;
  }
  100% {
    background-position: 0 0;
  }
}
```

Prepping Models and Animations

Now that we have a progress bar let's deal with the models. These models have animations and we want to be able to access those animations. Animations are stored in an array by default be we'd like to be able to easily access them by name so let's setup an animations property for each model to do that. Note of course this means animations must have unique names.

javascript

```
+function prepModelsAndAnimations() {
+  Object.values(models).forEach(model => {
+    const animsByName = {};
+    model.gltf.animations.forEach((clip) => {
+      animsByName[clip.name] = clip;
+    });
+    model.animations = animsByName;
+  });
+}

function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

+  prepModelsAndAnimations();
}
```

Displaying Animated Models with SkeletonUtils.clone

Let's display the animated models.

Unlike the previous example of loading a glTF file This time we probably want to be able to display more than one instance of each model. To do this, instead of adding the loaded gltf scene directly like we did in the article on loading a glTF, we instead want to clone the scene and in particular we want to clone it for skinned animated characters. Fortunately there's a utility function, SkeletonUtils.clone we can use to do this. So, first we need to include the utils.

Then we can clone the models we just loaded

Above, for each model, we clone the gltf.scene we loaded and we parent that to a new Object3D. We need to parent it to another object because when we play animations the animation will apply animated positions to the nodes in the loaded scene which means we won't have control over those positions.

javascript

```
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
import {GLTFLoader} from 'three/addons/loaders/GLTFLoader.js';
+import * as SkeletonUtils from 'three/addons/utils/SkeletonUtils.js';
```

javascript

```
function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

  prepModelsAndAnimations();

+  Object.values(models).forEach((model, ndx) => {
+    const clonedScene = SkeletonUtils.clone(model.gltf.scene);
+    const root = new THREE.Object3D();
+    root.add(clonedScene);
+    scene.add(root);
+    root.position.x = (ndx - 3) * 3;
+  });
}
```

Playing Animations with AnimationMixer

To play the animations each model we clone needs an AnimationMixer. An AnimationMixer contains 1 or more AnimationActions. An AnimationAction references an AnimationClip. AnimationActions have all kinds of settings for playing then chaining to another action or cross fading between actions. Let's just get the first AnimationClip and create an action for it. The default is for an action to play its clip in a loop forever.

We called play to start the action and stored off all the AnimationMixers in an array called mixers. Finally we need to update each AnimationMixer in our render loop by computing the time since the last frame and passing that to AnimationMixer.update.

And with that we should get each model loaded and playing its first animation.

javascript

```
+const mixers = [];

function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

  prepModelsAndAnimations();

  Object.values(models).forEach((model, ndx) => {
    const clonedScene = SkeletonUtils.clone(model.gltf.scene);
    const root = new THREE.Object3D();
    root.add(clonedScene);
    scene.add(root);
    root.position.x = (ndx - 3) * 3;

+    const mixer = new THREE.AnimationMixer(clonedScene);
+    const firstClip = Object.values(model.animations)[0];
+    const action = mixer.clipAction(firstClip);
+    action.play();
+    mixers.push(mixer);
  });
}
```

javascript

```
+let then = 0;
function render(now) {
+  now *= 0.001;  // convert to seconds
+  const deltaTime = now - then;
+  then = now;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

+  for (const mixer of mixers) {
+    mixer.update(deltaTime);
+  }

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

Cycling Through Animations

Let's make it so we can check all of the animations. We'll add all of the clips as actions and then enable just one at a time.

The code above makes an array of AnimationActions, one for each AnimationClip. It makes an array of objects, mixerInfos, with references to the AnimationMixer and all

the AnimationActions for each model. It then calls playNextAction which sets enabled on all but one action for that mixer.

We need to update the render loop for the new array

Let's make it so pressing a key 1 to 8 will play the next animation for each model

Now you should be able to click on the example and then press keys 1 through 8 to cycle each of the models through their available animations.

```javascript
```

```
-const mixers = [];
+const mixerInfos = [];

function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

  prepModelsAndAnimations();

  Object.values(models).forEach((model, ndx) => {
    const clonedScene = SkeletonUtils.clone(model.gltf.scene);
    const root = new THREE.Object3D();
    root.add(clonedScene);
    scene.add(root);
    root.position.x = (ndx - 3) * 3;

    const mixer = new THREE.AnimationMixer(clonedScene);
-    const firstClip = Object.values(model.animations)[0];
-    const action = mixer.clipAction(firstClip);
-    action.play();
-    mixers.push(mixer);
+    const actions = Object.values(model.animations).map((clip) => {
+      return mixer.clipAction(clip);
+    });
+    const mixerInfo = {
+      mixer,
+      actions,
+      actionNdx: -1,
+    };
+    mixerInfos.push(mixerInfo);
+    playNextAction(mixerInfo);
  });
}

+function playNextAction(mixerInfo) {
+  const {actions, actionNdx} = mixerInfo;
+  const nextActionNdx = (actionNdx + 1) % actions.length;
+  mixerInfo.actionNdx = nextActionNdx;
+  actions.forEach((action, ndx) => {
+    const enabled = ndx === nextActionNdx;
+    action.enabled = enabled;
+    if (enabled) {
+      action.play();
+    }
+  });
+}
```

javascript

```
-for (const mixer of mixers) {
+for (const {mixer} of mixerInfos) {
  mixer.update(deltaTime);
}
```

javascript

```
window.addEventListener('keydown', (e) => {
  const mixerInfo = mixerInfos[e.keyCode - 49];
  if (!mixerInfo) {
    return;
  }
  playNextAction(mixerInfo);
});
```

Entity Component System: GameObject and Component

So that is arguably the sum-total of the three.js portion of this article. We covered loading multiple files, cloning skinned models, and playing animations on them. In a real game you'd have to do a ton more manipulation of AnimationAction objects.

Let's start making a game infrastructure

A common pattern for making a modern game is to use an Entity Component System. In an Entity Component System an object in a game is called an entity that consists of a bunch of components. You build up entities by deciding which components to attach to them. So, let's make an Entity Component System.

We'll call our entities GameObject. It's effectively just a collection of components and a three.js Object3D.

Calling GameObject.update calls update on all the components.

I included a name only to help in debugging so if I look at a GameObject in the debugger I can see a name to help identify it.

Some things that might seem a little strange:

GameObject.addComponent is used to create components. Whether or not this a good idea or a bad idea I'm not sure. My thinking was it makes no sense for a component to exist outside of a gameobject so I thought it might be good if creating a component automatically added that component to the gameobject and passed the gameobject to the component's constructor. In other words to add a component you do this

If I didn't do it this way you'd instead do something like this

Is it better that the first way is shorter and more automated or is it worse because it looks out of the ordinary? I don't know.

GameObject.getComponent looks up components by type. That has the implication that you can not have 2 components of the same type on a single game object or at least if you do you can only look up the first one without adding some other API.

It's common for one component to look up another and when looking them up they have to match by type otherwise you might get the wrong one. We could instead give each component a name and you could look them up by name. That would be more flexible in that you could have more than one component of the same type but it would also be more tedious. Again, I'm not sure which is better.

On to the components themselves. Here is their base class.

Do components need a base class? JavaScript is not like most strictly typed languages so effectively we could have no base class and just leave it up to each component to do whatever it wants in its constructor knowing that the first argument is always the component's gameobject. If it doesn't care about gameobject it wouldn't store it. I kind of feel like this common base is good though. It means if you have a reference to a component you know you can find its parent gameobject always and from its parent you can easily look up other components as well as look at its transform.

```javascript
```

```
function removeArrayElement(array, element) {
  const ndx = array.indexOf(element);
  if (ndx >= 0) {
    array.splice(ndx, 1);
  }
}

class GameObject {
  constructor(parent, name) {
    this.name = name;
    this.components = [];
    this.transform = new THREE.Object3D();
    parent.add(this.transform);
  }
  addComponent(ComponentType, ...args) {
    const component = new ComponentType(this, ...args);
    this.components.push(component);
    return component;
  }
  removeComponent(component) {
    removeArrayElement(this.components, component);
  }
  getComponent(ComponentType) {
    return this.components.find(c => c instanceof ComponentType);
  }
  update() {
    for (const component of this.components) {
      component.update();
    }
  }
}
```

javascript

```
const gameObject = new GameObject(scene, 'foo');
gameObject.addComponent(TypeOfComponent);
```

javascript

```
const gameObject = new GameObject(scene, 'foo');
const component = new TypeOfComponent(gameObject);
gameObject.addComponent(component);
```

javascript

```
// Base for all components
class Component {
  constructor(gameObject) {
    this.gameObject = gameObject;
  }
  update() {
  }
}
```

GameObjectManager with SafeArray

To manage the gameobjects we probably need some kind of gameobject manager. You might think we could just keep an array of gameobjects but in a real game the components of a gameobject might add and remove other gameobjects at runtime. For example a gun gameobject might add a bullet gameobject every time the gun fires. A monster gameobject might remove itself if it has been killed. We then would have an issue that we might have code like this

The loop above would fail or do un-expected things if gameobjects are added or removed from globalArrayOfGameObjects in the middle of the loop in some component's update function.

To try to prevent that problem we need something a little safer. Here's one attempt.

The class above lets you add or remove elements from the SafeArray but won't mess with the array itself while it's being iterated over. Instead new elements get added to addQueue and removed elements to the removeQueue and then added or removed outside of the loop.

Using that here is our class to manage gameobjects.

javascript

```
for (const gameObject of globalArrayOfGameObjects) {
  gameObject.update();
}
```

javascript

```javascript
class SafeArray {
  constructor() {
    this.array = [];
    this.addQueue = [];
    this.removeQueue = new Set();
  }
  get isEmpty() {
    return this.addQueue.length + this.array.length > 0;
  }
  add(element) {
    this.addQueue.push(element);
  }
  remove(element) {
    this.removeQueue.add(element);
  }
  forEach(fn) {
    this._addQueued();
    this._removeQueued();
    for (const element of this.array) {
      if (this.removeQueue.has(element)) {
        continue;
      }
      fn(element);
    }
    this._removeQueued();
  }
  _addQueued() {
    if (this.addQueue.length) {
      this.array.splice(this.array.length, 0, ...this.addQueue);
      this.addQueue = [];
    }
  }
  _removeQueued() {
    if (this.removeQueue.size) {
      this.array = this.array.filter(element => !this.removeQueue.has(element));
      this.removeQueue.clear();
    }
  }
}
```

javascript

```
class GameObjectManager {
  constructor() {
    this.gameObjects = new SafeArray();
  }
  createGameObject(parent, name) {
    const gameObject = new GameObject(parent, name);
    this.gameObjects.add(gameObject);
    return gameObject;
  }
  removeGameObject(gameObject) {
    this.gameObjects.remove(gameObject);
  }
  update() {
    this.gameObjects.forEach(gameObject => gameObject.update());
  }
}
```

SkinInstance Component

With all that now let's make our first component. This component will just manage a skinned three.js object like the ones we just created. To keep it simple it will just have one method, setAnimation that takes the name of the animation to play and plays it.

You can see it's basically the code we had before that clones the scene we loaded, then sets up an AnimationMixer. setAnimation adds an AnimationAction for a particular AnimationClip if one does not already exist and disables all existing actions.

The code references globals.deltaTime. Let's make a globals object

And update it in the render loop

The check above for making sure deltaTime is not more than 1/20th of a second is because otherwise we'd get a huge value for deltaTime if we hide the tab. We might hide it for seconds or minutes and then when our tab was brought to the front deltaTime would be huge and might teleport characters across our game world if we had code like

By limiting the maximum deltaTime that issue is prevented.

```
javascript
```

```
class SkinInstance extends Component {
  constructor(gameObject, model) {
    super(gameObject);
    this.model = model;
    this.animRoot = SkeletonUtils.clone(this.model.gltf.scene);
    this.mixer = new THREE.AnimationMixer(this.animRoot);
    gameObject.transform.add(this.animRoot);
    this.actions = {};
  }
  setAnimation(animName) {
    const clip = this.model.animations[animName];
    // turn off all current actions
    for (const action of Object.values(this.actions)) {
      action.enabled = false;
    }
    // get or create existing action for clip
    const action = this.mixer.clipAction(clip);
    action.enabled = true;
    action.reset();
    action.play();
    this.actions[animName] = action;
  }
  update() {
    this.mixer.update(globals.deltaTime);
  }
}
```

javascript

```
const globals = {
  time: 0,
  deltaTime: 0,
};
```

javascript

```
let then = 0;
function render(now) {
  // convert to seconds
  globals.time = now * 0.001;
  // make sure delta time isn't too big.
  globals.deltaTime = Math.min(globals.time - then, 1 / 20);
  then = globals.time;
```

javascript

```
position += velocity * deltaTime;
```

Player Component and Animation Names

Now let's make a component for the player.

The player calls setAnimation with 'Run'. To know which animations are available I modified our previous example to print out the names of the animations

And running it got this list in the JavaScript console.

Fortunately the names of the animations for all the animals match which will come in handy later. For now we only care the that the player has an animation called Run.

Let's use these components. Here's the updated init function. All it does is create a GameObject and add a Player component to it.

And we need to call gameObjectManager.update in our render loop

and if we run that we get a single player.

That was a lot of code just for an entity component system but it's infrastructure that most games need.

javascript

```
class Player extends Component {
  constructor(gameObject) {
    super(gameObject);
    const model = models.knight;
    this.skinInstance = gameObject.addComponent(SkinInstance, model);
    this.skinInstance.setAnimation('Run');
  }
}
```

javascript

```
function prepModelsAndAnimations() {
  Object.values(models).forEach(model => {
+    console.log('------->:', model.url);
    const animsByName = {};
    model.gltf.animations.forEach((clip) => {
      animsByName[clip.name] = clip;
+      console.log('  ', clip.name);
    });
    model.animations = animsByName;
  });
}
```

javascript

```
-------->:  resources/models/animals/Pig.gltf
    Idle
    Death
    WalkSlow
    Jump
    Walk
 -------->:  resources/models/animals/Cow.gltf
    Walk
    Jump
    WalkSlow
    Death
    Idle
 -------->:  resources/models/animals/Llama.gltf
    Jump
    Idle
    Walk
    Death
    WalkSlow
 -------->:  resources/models/animals/Pug.gltf
    Jump
    Walk
    Idle
    WalkSlow
    Death
 -------->:  resources/models/animals/Sheep.gltf
    WalkSlow
    Death
    Jump
    Walk
    Idle
 -------->:  resources/models/animals/Zebra.gltf
    Jump
    Walk
    Death
    WalkSlow
    Idle
 -------->:  resources/models/animals/Horse.gltf
    Jump
    WalkSlow
    Death
    Walk
    Idle
 -------->:  resources/models/knight/KnightCharacter.gltf
    Run_swordRight
    Run
    Idle_swordLeft
    Roll_sword
    Idle
    Run_swordAttack
```

```
javascript
```

```
const globals = {
  time: 0,
  deltaTime: 0,
};
+const gameObjectManager = new GameObjectManager();

function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

  prepModelsAndAnimations();

+  {
+    const gameObject = gameObjectManager.createGameObject(scene, 'player');
+    gameObject.addComponent(Player);
+  }
}
```

javascript

```
let then = 0;
function render(now) {
  // convert to seconds
  globals.time = now * 0.001;
  // make sure delta time isn't too big.
  globals.deltaTime = Math.min(globals.time - then, 1 / 20);
  then = globals.time;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

-  for (const {mixer} of mixerInfos) {
-    mixer.update(deltaTime);
-  }
+  gameObjectManager.update();

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}
```

Input System

Let's add an input system. Rather than read keys directly we'll make a class that other parts of the code can check left or right. That way we can assign multiple ways to input left or right etc.. We'll start with just keys

The code above tracks whether keys are up or down and you can check if a key is currently pressed by checking for example inputManager.keys.left.down. It also has a

justPressed property for each key so that you can check the user just pressed the key. For example a jump key you don't want to know if the button is being held down, you want to know did the user press it now.

Let's create an instance of InputManager

and update it in our render loop

It needs to be called after gameObjectManager.update otherwise justPressed would never be true inside a component's update function.

Let's use it in the Player component

The code above uses Object3D.transformOnAxis to move the player forward. Object3D.transformOnAxis works in local space so it only works if the object in question is at the root of the scene, not if it's parented to something else.

We also added a global moveSpeed and based a turnSpeed on the move speed. The turn speed is based on the move speed to try to make sure a character can turn sharply enough to meet its target. If turnSpeed so too small a character will turn around and around circling its target but never hitting it. I didn't bother to do the math to calculate the required turn speed for a given move speed. I just guessed.

```javascript
```

```
// Keeps the state of keys/buttons
//
// You can check
//
//   inputManager.keys.left.down
//
// to see if the left key is currently held down
// and you can check
//
//   inputManager.keys.left.justPressed
//
// To see if the left key was pressed this frame
//
// Keys are 'left', 'right', 'a', 'b', 'up', 'down'
class InputManager {
  constructor() {
    this.keys = {};
    const keyMap = new Map();

    const setKey = (keyName, pressed) => {
      const keyState = this.keys[keyName];
      keyState.justPressed = pressed && !keyState.down;
      keyState.down = pressed;
    };

    const addKey = (keyCode, name) => {
      this.keys[name] = { down: false, justPressed: false };
      keyMap.set(keyCode, name);
    };

    const setKeyFromKeyCode = (keyCode, pressed) => {
      const keyName = keyMap.get(keyCode);
      if (!keyName) {
        return;
      }
      setKey(keyName, pressed);
    };

    addKey(37, 'left');
    addKey(39, 'right');
    addKey(38, 'up');
    addKey(40, 'down');
    addKey(90, 'a');
    addKey(88, 'b');

    window.addEventListener('keydown', (e) => {
      setKeyFromKeyCode(e.keyCode, true);
    });
    window.addEventListener('keyup', (e) => {
      setKeyFromKeyCode(e.keyCode, false);
    });
  }
  update() {
    for (const keyState of Object.values(this.keys)) {
      if (keyState.justPressed) {
        keyState.justPressed = false;
      }
    }
  }
}
```

javascript

```
const globals = {
  time: 0,
  deltaTime: 0,
};
const gameObjectManager = new GameObjectManager();
+const inputManager = new InputManager();
```

javascript

```
function render(now) {

  ...

  gameObjectManager.update();
+  inputManager.update();

  ...
}
```

javascript

```
+const kForward = new THREE.Vector3(0, 0, 1);
const globals = {
  time: 0,
  deltaTime: 0,
+  moveSpeed: 16,
};

class Player extends Component {
  constructor(gameObject) {
    super(gameObject);
    const model = models.knight;
    this.skinInstance = gameObject.addComponent(SkinInstance, model);
    this.skinInstance.setAnimation('Run');
+    this.turnSpeed = globals.moveSpeed / 4;
  }
+  update() {
+    const {deltaTime, moveSpeed} = globals;
+    const {transform} = this.gameObject;
+    const delta = (inputManager.keys.left.down  ?  1 : 0) +
+                  (inputManager.keys.right.down ? -1 : 0);
+    transform.rotation.y += this.turnSpeed * delta * deltaTime;
+    transform.translateOnAxis(kForward, moveSpeed * deltaTime);
+  }
}
```

Camera Frustum Check (Offscreen Detection)

The code so far would work but if the player runs off the screen there's no way to find out where they are. Let's make it so if they are offscreen for more than a certain time

they get teleported back to the origin. We can do that by using the three.js Frustum class to check if a point is inside the camera's view frustum.

We need to build a frustum from the camera. We could do this in the Player component but other objects might want to use this too so let's add another gameobject with a component to manage a frustum.

Then let's setup another gameobject at init time.

and now we can use it in the Player component.

javascript

```
class CameraInfo extends Component {
  constructor(gameObject) {
    super(gameObject);
    this.projScreenMatrix = new THREE.Matrix4();
    this.frustum = new THREE.Frustum();
  }
  update() {
    const {camera} = globals;
    this.projScreenMatrix.multiplyMatrices(
        camera.projectionMatrix,
        camera.matrixWorldInverse);
    this.frustum.setFromProjectionMatrix(this.projScreenMatrix);
  }
}
```

javascript

```
function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

  prepModelsAndAnimations();

+  {
+    const gameObject = gameObjectManager.createGameObject(camera, 'camera');
+    globals.cameraInfo = gameObject.addComponent(CameraInfo);
+  }

  {
    const gameObject = gameObjectManager.createGameObject(scene, 'player');
    gameObject.addComponent(Player);
  }
}
```

javascript

```
class Player extends Component {
  constructor(gameObject) {
    super(gameObject);
    const model = models.knight;
    this.skinInstance = gameObject.addComponent(SkinInstance, model);
    this.skinInstance.setAnimation('Run');
    this.turnSpeed = globals.moveSpeed / 4;
+    this.offscreenTimer = 0;
+    this.maxTimeOffScreen = 3;
  }
  update() {
-    const {deltaTime, moveSpeed} = globals;
+    const {deltaTime, moveSpeed, cameraInfo} = globals;
    const {transform} = this.gameObject;
    const delta = (inputManager.keys.left.down  ?  1 : 0) +
                  (inputManager.keys.right.down ? -1 : 0);
    transform.rotation.y += this.turnSpeed * delta * deltaTime;
    transform.translateOnAxis(kForward, moveSpeed * deltaTime);

+    const {frustum} = cameraInfo;
+    if (frustum.containsPoint(transform.position)) {
+      this.offscreenTimer = 0;
+    } else {
+      this.offscreenTimer += deltaTime;
+      if (this.offscreenTimer >= this.maxTimeOffScreen) {
+        transform.position.set(0, 0, 0);
+      }
+    }
  }
}
```

Touchscreen Support

One more thing before we try it out, let's add touchscreen support for mobile. First let's add some HTML to touch

and some CSS to style it

The idea here is there is one div, #ui, that covers the entire page. Inside will be 2 divs, #left and #right both of which are almost half the page wide and the entire screen tall. In between there is a 40px separator. If the user slides their finger over the left or right side then we need up update keys.left and keys.right in the InputManager. This makes the entire screen sensitive to being touched which seemed better than just small arrows.

And now we should be able to control the character with the left and right cursor keys or with our fingers on a touchscreen.

Ideally we'd do something else if the player went off the screen like move the camera or maybe offscreen = death but this article is already going to be too long so for now

teleporting to the middle was the simplest thing.

html

```
<body>
  <canvas id="c"></canvas>
+  <div id="ui">
+    <div id="left"><img src="../resources/images/left.svg"></div>
+    <div style="flex: 0 0 40px;"></div>
+    <div id="right"><img src="../resources/images/right.svg"></div>
+  </div>
  <div id="loading">
    <div>
      <div>...loading...</div>
      <div class="progress"><div id="progressbar"></div></div>
    </div>
  </div>
</body>
```

css

```
#ui {
  position: absolute;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  display: flex;
  justify-items: center;
  align-content: stretch;
}
#ui>div {
  display: flex;
  align-items: flex-end;
  flex: 1 1 auto;
}
.bright {
  filter: brightness(2);
}
#left {
  justify-content: flex-end;
}
#right {
  justify-content: flex-start;
}
#ui img {
  padding: 10px;
  width: 80px;
  height: 80px;
  display: block;
}
```

javascript

```
class InputManager {
  constructor() {
    this.keys = {};
    const keyMap = new Map();

    const setKey = (keyName, pressed) => {
      const keyState = this.keys[keyName];
      keyState.justPressed = pressed && !keyState.down;
      keyState.down = pressed;
    };

    const addKey = (keyCode, name) => {
      this.keys[name] = { down: false, justPressed: false };
      keyMap.set(keyCode, name);
    };

    const setKeyFromKeyCode = (keyCode, pressed) => {
      const keyName = keyMap.get(keyCode);
      if (!keyName) {
        return;
      }
      setKey(keyName, pressed);
    };

    addKey(37, 'left');
    addKey(39, 'right');
    addKey(38, 'up');
    addKey(40, 'down');
    addKey(90, 'a');
    addKey(88, 'b');

    window.addEventListener('keydown', (e) => {
      setKeyFromKeyCode(e.keyCode, true);
    });
    window.addEventListener('keyup', (e) => {
      setKeyFromKeyCode(e.keyCode, false);
    });

+    const sides = [
+      { elem: document.querySelector('#left'),  key: 'left'  },
+      { elem: document.querySelector('#right'), key: 'right' },
+    ];
+
+    const clearKeys = () => {
+      for (const {key} of sides) {
+          setKey(key, false);
+      }
+    };
+
+    const handleMouseMove = (e) => {
+      e.preventDefault();
+      // this is needed because we call preventDefault();
+      // we also gave the canvas a tabindex so it can
+      // become the focus
+      canvas.focus();
+      window.addEventListener('pointermove', handleMouseMove);
+      window.addEventListener('pointerup', handleMouseUp);
+
+      for (const {elem, key} of sides) {
+        let pressed = false;
+        const rect = elem.getBoundingClientRect();
```

```
+        const x = e.clientX;
+        const y = e.clientY;
+        const inRect = x >= rect.left && x < rect.right &&
+                       y >= rect.top && y < rect.bottom;
+        if (inRect) {
+          pressed = true;
+        }
+        setKey(key, pressed);
+      }
+    };
+
+    function handleMouseUp() {
+      clearKeys();
+      window.removeEventListener('pointermove', handleMouseMove, {passive: false})
+      window.removeEventListener('pointerup', handleMouseUp);
+    }
+
+    const uiElem = document.querySelector('#ui');
+    uiElem.addEventListener('pointerdown', handleMouseMove, {passive: false});
+
+    uiElem.addEventListener('touchstart', (e) => {
+      // prevent scrolling
+      e.preventDefault();
+    }, {passive: false});
  }
  update() {
    for (const keyState of Object.values(this.keys)) {
      if (keyState.justPressed) {
        keyState.justPressed = false;
      }
    }
  }
}
```

Animal Component Setup

Lets add some animals. We can start it off similar to the Player by making an Animal component.

The code above sets the AnimationMixer.timeScale to set the playback speed of the animations relative to the move speed. This way if we adjust the move speed the animation will speed up or slow down as well.

To start we could setup one of each type of animal

And that would get us animals standing on the screen but we want them to do something.

```
javascript
```

```
class Animal extends Component {
  constructor(gameObject, model) {
    super(gameObject);
    const skinInstance = gameObject.addComponent(SkinInstance, model);
    skinInstance.mixer.timeScale = globals.moveSpeed / 4;
    skinInstance.setAnimation('Idle');
  }
}
```

javascript

```
function init() {
  // hide the loading bar
  const loadingElem = document.querySelector('#loading');
  loadingElem.style.display = 'none';

  prepModelsAndAnimations();
  {
    const gameObject = gameObjectManager.createGameObject(camera, 'camera');
    globals.cameraInfo = gameObject.addComponent(CameraInfo);
  }

  {
    const gameObject = gameObjectManager.createGameObject(scene, 'player');
    globals.player = gameObject.addComponent(Player);
    globals.congaLine = [gameObject];
  }

+  const animalModelNames = [
+    'pig',
+    'cow',
+    'llama',
+    'pug',
+    'sheep',
+    'zebra',
+    'horse',
+  ];
+  animalModelNames.forEach((name, ndx) => {
+    const gameObject = gameObjectManager.createGameObject(scene, name);
+    gameObject.addComponent(Animal, models[name]);
+    gameObject.transform.position.x = (ndx + 1) * 5;
+  });
}
```

Animal AI States (Conga Line)

Let's make them follow the player in a conga line but only if the player gets near enough. To do this we need several states.

- Idle: Animal is waiting for player to get close
- Wait for End of Line: Animal was tagged by player but now needs to wait for the animal at the end of the line to come by so they can join the end of the line.

- Go to Last: Animal needs to walk to where the animal they are following was, at the same time recording a history of where the animal they are following is currently.
- Follow: Animal needs to keep recording a history of where the animal they are following is while moving to where the animal they are following was before.

There are many ways to handle different states like this. A common one is to use a Finite State Machine and to build some class to help us manage the state.

So, let's do that.

Here's a simple class. We pass it an object with a bunch of states. Each state as 3 optional functions, enter, update, and exit. To switch states we call FiniteStateMachine.transition and pass it the name of the new state. If the current state has an exit function it's called. Then if the new state has an enter function it's called. Finally each frame FiniteStateMachine.update calls the update function of the current state.

Let's use it to manage the states of the animals.

That was big chunk of code but it does what was described above. Hopefully of you walk through each state it will be clear.

A few things we need to add. We need the player to add itself to the globals so the animals can find it and we need to start the conga line with the player's GameObject.

We also need to compute a size for each model

And we need the player to record their size

Thinking about it now it would probably have been smarter for the animals to just target the head of the conga line instead of the player specifically. Maybe I'll come back and change that later.

When I first started this I used just one radius for all animals but of course that was no good as the pug is much smaller than the horse. So I added the difference sizes but I wanted to be able to visualize things. To do that I made a StateDisplayHelper component.

I uses a PolarGridHelper to draw a circle around each character and it uses html elements to let each character show some status using the techniques covered in the article on aligning html elements to 3D.

First we need to add some HTML to host these elements

And add some CSS for them

javascript

```
class FiniteStateMachine {
  constructor(states, initialState) {
    this.states = states;
    this.transition(initialState);
  }
  get state() {
    return this.currentState;
  }
  transition(state) {
    const oldState = this.states[this.currentState];
    if (oldState && oldState.exit) {
      oldState.exit.call(this);
    }
    this.currentState = state;
    const newState = this.states[state];
    if (newState.enter) {
      newState.enter.call(this);
    }
  }
  update() {
    const state = this.states[this.currentState];
    if (state.update) {
      state.update.call(this);
    }
  }
}
```

javascript

```
// Returns true of obj1 and obj2 are close
function isClose(obj1, obj1Radius, obj2, obj2Radius) {
  const minDist = obj1Radius + obj2Radius;
  const dist = obj1.position.distanceTo(obj2.position);
  return dist < minDist;
}

// keeps v between -min and +min
function minMagnitude(v, min) {
  return Math.abs(v) > min
      ? min * Math.sign(v)
      : v;
}

const aimTowardAndGetDistance = function() {
  const delta = new THREE.Vector3();

  return function aimTowardAndGetDistance(source, targetPos, maxTurn) {
    delta.subVectors(targetPos, source.position);
    // compute the direction we want to be facing
    const targetRot = Math.atan2(delta.x, delta.z) + Math.PI * 1.5;
    // rotate in the shortest direction
    const deltaRot = (targetRot - source.rotation.y + Math.PI * 1.5) % (Math.PI * 2
    // make sure we don't turn faster than maxTurn
    const deltaRotation = minMagnitude(deltaRot, maxTurn);
    // keep rotation between 0 and Math.PI * 2
    source.rotation.y = THREE.MathUtils.euclideanModulo(
        source.rotation.y + deltaRotation, Math.PI * 2);
    // return the distance to the target
    return delta.length();
  };
}();

class Animal extends Component {
  constructor(gameObject, model) {
    super(gameObject);
+    const hitRadius = model.size / 2;
    const skinInstance = gameObject.addComponent(SkinInstance, model);
    skinInstance.mixer.timeScale = globals.moveSpeed / 4;
+    const transform = gameObject.transform;
+    const playerTransform = globals.player.gameObject.transform;
+    const maxTurnSpeed = Math.PI * (globals.moveSpeed / 4);
+    const targetHistory = [];
+    let targetNdx = 0;
+
+    function addHistory() {
+      const targetGO = globals.congaLine[targetNdx];
+      const newTargetPos = new THREE.Vector3();
+      newTargetPos.copy(targetGO.transform.position);
+      targetHistory.push(newTargetPos);
+    }
+
+    this.fsm = new FiniteStateMachine({
+      idle: {
+        enter: () => {
+          skinInstance.setAnimation('Idle');
+        },
+        update: () => {
+          // check if player is near
+          if (isClose(transform, hitRadius, playerTransform, globals.playerRadius)
+            this.fsm.transition('waitForEnd');
```

```
+          }
+        },
+      },
+      waitForEnd: {
+        enter: () => {
+          skinInstance.setAnimation('Jump');
+        },
+        update: () => {
+          // get the gameObject at the end of the conga line
+          const lastGO = globals.congaLine[globals.congaLine.length - 1];
+          const deltaTurnSpeed = maxTurnSpeed * globals.deltaTime;
+          const targetPos = lastGO.transform.position;
+          aimTowardAndGetDistance(transform, targetPos, deltaTurnSpeed);
+          // check if last thing in conga line is near
+          if (isClose(transform, hitRadius, lastGO.transform, globals.playerRadius
+            this.fsm.transition('goToLast');
+          }
+        },
+      },
+      goToLast: {
+        enter: () => {
+          // remember who we're following
+          targetNdx = globals.congaLine.length - 1;
+          // add ourselves to the conga line
+          globals.congaLine.push(gameObject);
+          skinInstance.setAnimation('Walk');
+        },
+        update: () => {
+          addHistory();
+          // walk to the oldest point in the history
+          const targetPos = targetHistory[0];
+          const maxVelocity = globals.moveSpeed * globals.deltaTime;
+          const deltaTurnSpeed = maxTurnSpeed * globals.deltaTime;
+          const distance = aimTowardAndGetDistance(transform, targetPos, deltaTurn
+          const velocity = distance;
+          transform.translateOnAxis(kForward, Math.min(velocity, maxVelocity));
+          if (distance <= maxVelocity) {
+            this.fsm.transition('follow');
+          }
+        },
+      },
+      follow: {
+        update: () => {
+          addHistory();
+          // remove the oldest history and just put ourselves there.
+          const targetPos = targetHistory.shift();
+          transform.position.copy(targetPos);
+          const deltaTurnSpeed = maxTurnSpeed * globals.deltaTime;
+          aimTowardAndGetDistance(transform, targetHistory[0], deltaTurnSpeed);
+        },
+      },
+    }, 'idle');
+  }
+  update() {
+    this.fsm.update();
+  }
}
```

javascript

```
function init() {

  ...

  {
    const gameObject = gameObjectManager.createGameObject(scene, 'player');
+    globals.player = gameObject.addComponent(Player);
+    globals.congaLine = [gameObject];
  }

}
```

javascript

```
function prepModelsAndAnimations() {
+  const box = new THREE.Box3();
+  const size = new THREE.Vector3();
  Object.values(models).forEach(model => {
+    box.setFromObject(model.gltf.scene);
+    box.getSize(size);
+    model.size = size.length();
    const animsByName = {};
    model.gltf.animations.forEach((clip) => {
      animsByName[clip.name] = clip;
      // Should really fix this in .blend file
      if (clip.name === 'Walk') {
        clip.duration /= 2;
      }
    });
    model.animations = animsByName;
  });
}
```

javascript

```
class Player extends Component {
  constructor(gameObject) {
    super(gameObject);
    const model = models.knight;
+    globals.playerRadius = model.size / 2;
```

html

```
<body>
  <canvas id="c"></canvas>
  <div id="ui">
    <div id="left"><img src="../resources/images/left.svg"></div>
    <div style="flex: 0 0 40px;"></div>
    <div id="right"><img src="../resources/images/right.svg"></div>
  </div>
  <div id="loading">
    <div>
      <div>...loading...</div>
      <div class="progress"><div id="progressbar"></div></div>
    </div>
  </div>
+  <div id="labels"></div>
</body>
```

Voxel(Minecraft Like) Geometry

GUIDE

Source: https://threejs.org/manual/en/voxel-geometry.html

A tutorial on how to efficiently generate voxel-based geometry in Three.js, similar to Minecraft, by avoiding the common mistake of creating one mesh per voxel and instead building custom BufferGeometry with face culling, cells, textures, and editing support.

Introduction

I've seen this topic come up more than once in various places. That is basically, "How do I make a voxel display like Minecraft".

Most people first attempt this by making a cube geometry and then making a mesh at each voxel position. Just for fun I tried this. I made a 16777216 element `Uint8Array` to represent a 256x256x256 cube of voxels.

javascript

```
const cellSize = 256;
const cell = new Uint8Array(cellSize * cellSize * cellSize);
```

Filling Voxels with Hills

I then made a single layer with a kind of hills of sine waves like this

javascript

```
for (let y = 0; y < cellSize; ++y) {
  for (let z = 0; z < cellSize; ++z) {
    for (let x = 0; x < cellSize; ++x) {
      const height = (Math.sin(x / cellSize * Math.PI * 4) + Math.sin(z / cellSize
      if (height > y && height < y + 1) {
        const offset = y * cellSize * cellSize +
                       z * cellSize +
                       x;
        cell[offset] = 1;
      }
    }
  }
}
```

Creating a Mesh per Voxel

I then walked through all the cells and if they were not 0 I created a mesh with a cube.

javascript

```
const geometry = new THREE.BoxGeometry(1, 1, 1);
const material = new THREE.MeshPhongMaterial({color: 'green'});

for (let y = 0; y < cellSize; ++y) {
  for (let z = 0; z < cellSize; ++z) {
    for (let x = 0; x < cellSize; ++x) {
      const offset = y * cellSize * cellSize +
                     z * cellSize +
                     x;
      const block = cell[offset];
      const mesh = new THREE.Mesh(geometry, material);
      mesh.position.set(x, y, z);
      scene.add(mesh);
    }
  }
}
```

Performance Issues

The rest of the code is based on the example from the article on rendering on demand.

click here to open in a separate window

It takes a while to start and if you try to move the camera it's likely too slow. Like the article on how to optimize lots of objects the problem is there are just way too many objects. 256x256 is 65536 boxes!

Using the technique of merging the geometry will fix the issue for this example but what if instead of just making a single layer we filled in everything below the ground

with voxel. In other words change the loop filling in the voxels to this

```javascript
 for (let y = 0; y < cellSize; ++y) {
   for (let z = 0; z < cellSize; ++z) {
     for (let x = 0; x < cellSize; ++x) {
       const height = (Math.sin(x / cellSize * Math.PI * 4) + Math.sin(z / cellSize
-       if (height > y && height < y + 1) {
+       if (height < y + 1) {
         const offset = y * cellSize * cellSize +
                        z * cellSize +
                        x;
         cell[offset] = 1;
       }
     }
   }
 }
```

Memory and Face Culling

I tried it once just to see the results. It churned for about a minute and then crashed with *out of memory* 😁

There are several issues but the biggest issue is we're making all these faces inside the cubes that we can actually never see.

In other words lets say we have a box of voxels 3x2x2. By merging cubes we're getting this

but we really want this

In the top box there are faces between the voxels. Faces that are a waste since they can't be seen. It's not just one face between each voxel, there are 2 faces, one for each voxel facing its neighbor that are a waste. All these extra faces, especially for a large volume of voxels will kill performance.

It should be clear that we can't just merge geometry. We need to build it ourselves, taking into account that if a voxel has an adjacent neighbor it doesn't need the face facing that neighbor.

The next issue is that 256x256x256 is just too big. 16meg is a lot of memory and if nothing else in much of the space nothing is there so that's a lot of wasted memory. It's also a huge number of voxels, 16 million! That's too much to consider at once.

A solution is to divide the area into smaller areas. Any area that has nothing in it needs no storage. Let's use 32x32x32 areas (that's 32k) and only create an area if something is in it. We'll call one of these larger 32x32x32 areas a "cell".

Let's break this into pieces. First let's make a class to manage the voxel data.

javascript

```
class VoxelWorld {
  constructor(cellSize) {
    this.cellSize = cellSize;
  }
}
```

Generating Geometry Data for a Cell

Let's make the function that makes geometry for a cell. Let's assume you pass in a cell position. In other words if you want the geometry for the cell that covers voxels (0-31x, 0-31y, 0-31z) then you'd pass in 0,0,0. For the cell that covers voxels (32-63x, 0-31y, 0-31z) you'd pass in 1,0,0.

We need to be able to check the neighboring voxels so let's assume our class has a function `getVoxel` that given a voxel position returns the value of the voxel there. In other words if you pass it 35,0,0 and the cellSize is 32 it's going to look at cell 1,0,0 and in that cell it will look at voxel 3,0,0. Using this function we can look at a voxel's neighboring voxels even if they happen to be in neighboring cells.

javascript

```
class VoxelWorld {
  constructor(cellSize) {
    this.cellSize = cellSize;
  }
+  generateGeometryDataForCell(cellX, cellY, cellZ) {
+    const {cellSize} = this;
+    const startX = cellX * cellSize;
+    const startY = cellY * cellSize;
+    const startZ = cellZ * cellSize;
+
+    for (let y = 0; y < cellSize; ++y) {
+      const voxelY = startY + y;
+      for (let z = 0; z < cellSize; ++z) {
+        const voxelZ = startZ + z;
+        for (let x = 0; x < cellSize; ++x) {
+          const voxelX = startX + x;
+          const voxel = this.getVoxel(voxelX, voxelY, voxelZ);
+          if (voxel) {
+            for (const {dir} of VoxelWorld.faces) {
+              const neighbor = this.getVoxel(
+                  voxelX + dir[0],
+                  voxelY + dir[1],
+                  voxelZ + dir[2]);
+              if (!neighbor) {
+                // this voxel has no neighbor in this direction so we need a face
+                // here.
+              }
+            }
+          }
+        }
+      }
+    }
+  }
}

+VoxelWorld.faces = [
+  { // left
+    dir: [ -1,  0,  0, ],
+  },
+  { // right
+    dir: [  1,  0,  0, ],
+  },
+  { // bottom
+    dir: [  0, -1,  0, ],
+  },
+  { // top
+    dir: [  0,  1,  0, ],
+  },
+  { // back
+    dir: [  0,  0, -1, ],
+  },
+  { // front
+    dir: [  0,  0,  1, ],
+  },
+];
```

Building the Faces with Positions, Normals, and Indices

So using the code above we know when we need a face. Let's generate the faces.

```javascript
```

```
class VoxelWorld {
  constructor(cellSize) {
    this.cellSize = cellSize;
  }
  generateGeometryDataForCell(cellX, cellY, cellZ) {
    const {cellSize} = this;
+    const positions = [];
+    const normals = [];
+    const indices = [];
    const startX = cellX * cellSize;
    const startY = cellY * cellSize;
    const startZ = cellZ * cellSize;

    for (let y = 0; y < cellSize; ++y) {
      const voxelY = startY + y;
      for (let z = 0; z < cellSize; ++z) {
        const voxelZ = startZ + z;
        for (let x = 0; x < cellSize; ++x) {
          const voxelX = startX + x;
          const voxel = this.getVoxel(voxelX, voxelY, voxelZ);
          if (voxel) {
-            for (const {dir} of VoxelWorld.faces) {
+            for (const {dir, corners} of VoxelWorld.faces) {
+             const neighbor = this.getVoxel(
+                 voxelX + dir[0],
+                 voxelY + dir[1],
+                 voxelZ + dir[2]);
+             if (!neighbor) {
+               // this voxel has no neighbor in this direction so we need a face.
+               const ndx = positions.length / 3;
+               for (const pos of corners) {
+                 positions.push(pos[0] + x, pos[1] + y, pos[2] + z);
+                 normals.push(...dir);
+               }
+               indices.push(
+                 ndx, ndx + 1, ndx + 2,
+                 ndx + 2, ndx + 1, ndx + 3,
+               );
+             }
            }
          }
        }
      }
    }
+    return {
+      positions,
+      normals,
+      indices,
    };
  }
}

VoxelWorld.faces = [
  { // left
    dir: [ -1,  0,  0, ],
+    corners: [
+      [ 0, 1, 0 ],
+      [ 0, 0, 0 ],
+      [ 0, 1, 1 ],
+      [ 0, 0, 1 ],
+    ],
```

```
  },
  { // right
    dir: [  1,  0,  0, ],
+    corners: [
+      [ 1, 1, 1 ],
+      [ 1, 0, 1 ],
+      [ 1, 1, 0 ],
+      [ 1, 0, 0 ],
+    ],
  },
  { // bottom
    dir: [  0, -1,  0, ],
+    corners: [
+      [ 1, 0, 1 ],
+      [ 0, 0, 1 ],
+      [ 1, 0, 0 ],
+      [ 0, 0, 0 ],
+    ],
  },
  { // top
    dir: [  0,  1,  0, ],
+    corners: [
+      [ 0, 1, 1 ],
+      [ 1, 1, 1 ],
+      [ 0, 1, 0 ],
+      [ 1, 1, 0 ],
+    ],
  },
  { // back
    dir: [  0,  0, -1, ],
+    corners: [
+      [ 1, 0, 0 ],
+      [ 0, 0, 0 ],
+      [ 1, 1, 0 ],
+      [ 0, 1, 0 ],
+    ],
  },
  { // front
    dir: [  0,  0,  1, ],
+    corners: [
+      [ 0, 0, 1 ],
+      [ 1, 0, 1 ],
+      [ 0, 1, 1 ],
+      [ 1, 1, 1 ],
+    ],
  },
];
```

Supplying the getVoxel Function

The code above would make basic geometry data for us. We just need to supply the `getVoxel` function. Let's start with just one hard coded cell.

```
javascript
```

```
class VoxelWorld {
  constructor(cellSize) {
    this.cellSize = cellSize;
+    this.cell = new Uint8Array(cellSize * cellSize * cellSize);
  }
+  getCellForVoxel(x, y, z) {
+    const {cellSize} = this;
+    const cellX = Math.floor(x / cellSize);
+    const cellY = Math.floor(y / cellSize);
+    const cellZ = Math.floor(z / cellSize);
+    if (cellX !== 0 || cellY !== 0 || cellZ !== 0) {
+      return null
+    }
+    return this.cell;
+  }
+  getVoxel(x, y, z) {
+    const cell = this.getCellForVoxel(x, y, z);
+    if (!cell) {
+      return 0;
+    }
+    const {cellSize} = this;
+    const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
+    const voxelY = THREE.MathUtils.euclideanModulo(y, cellSize) | 0;
+    const voxelZ = THREE.MathUtils.euclideanModulo(z, cellSize) | 0;
+    const voxelOffset = voxelY * cellSize * cellSize +
+                        voxelZ * cellSize +
+                        voxelX;
+    return cell[voxelOffset];
+  }
  generateGeometryDataForCell(cellX, cellY, cellZ) {

  ...
}
```

Adding setVoxel

This seems like it would work. Let's make a `setVoxel` function so we can set some data.

```
javascript
```

```
class VoxelWorld {
  constructor(cellSize) {
    this.cellSize = cellSize;
    this.cell = new Uint8Array(cellSize * cellSize * cellSize);
  }
  getCellForVoxel(x, y, z) {
    const {cellSize} = this;
    const cellX = Math.floor(x / cellSize);
    const cellY = Math.floor(y / cellSize);
    const cellZ = Math.floor(z / cellSize);
    if (cellX !== 0 || cellY !== 0 || cellZ !== 0) {
      return null
    }
    return this.cell;
  }
+  setVoxel(x, y, z, v) {
+    let cell = this.getCellForVoxel(x, y, z);
+    if (!cell) {
+      return;  // TODO: add a new cell?
+    }
+    const {cellSize} = this;
+    const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
+    const voxelY = THREE.MathUtils.euclideanModulo(y, cellSize) | 0;
+    const voxelZ = THREE.MathUtils.euclideanModulo(z, cellSize) | 0;
+    const voxelOffset = voxelY * cellSize * cellSize +
+                        voxelZ * cellSize +
+                        voxelX;
+    cell[voxelOffset] = v;
+  }
  getVoxel(x, y, z) {
    const cell = this.getCellForVoxel(x, y, z);
    if (!cell) {
      return 0;
    }
    const {cellSize} = this;
    const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
    const voxelY = THREE.MathUtils.euclideanModulo(y, cellSize) | 0;
    const voxelZ = THREE.MathUtils.euclideanModulo(z, cellSize) | 0;
    const voxelOffset = voxelY * cellSize * cellSize +
                        voxelZ * cellSize +
                        voxelX;
    return cell[voxelOffset];
  }
  generateGeometryDataForCell(cellX, cellY, cellZ) {

  ...
}
```

Refactoring with computeVoxelOffset

Hmmm, I see a lot of repeated code. Let's fix that up

```
javascript
```

```
class VoxelWorld {
  constructor(cellSize) {
    this.cellSize = cellSize;
+    this.cellSliceSize = cellSize * cellSize;
    this.cell = new Uint8Array(cellSize * cellSize * cellSize);
  }
  getCellForVoxel(x, y, z) {
    const {cellSize} = this;
    const cellX = Math.floor(x / cellSize);
    const cellY = Math.floor(y / cellSize);
    const cellZ = Math.floor(z / cellSize);
    if (cellX !== 0 || cellY !== 0 || cellZ !== 0) {
      return null;
    }
    return this.cell;
  }
+  computeVoxelOffset(x, y, z) {
+    const {cellSize, cellSliceSize} = this;
+    const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
+    const voxelY = THREE.MathUtils.euclideanModulo(y, cellSize) | 0;
+    const voxelZ = THREE.MathUtils.euclideanModulo(z, cellSize) | 0;
+    return voxelY * cellSliceSize +
+           voxelZ * cellSize +
+           voxelX;
+  }
  setVoxel(x, y, z, v) {
    const cell = this.getCellForVoxel(x, y, z);
    if (!cell) {
      return;  // TODO: add a new cell?
    }
-    const {cellSize} = this;
-    const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
-    const voxelY = THREE.MathUtils.euclideanModulo(y, cellSize) | 0;
-    const voxelZ = THREE.MathUtils.euclideanModulo(z, cellSize) | 0;
-    const voxelOffset = voxelY * cellSize * cellSize +
-                        voxelZ * cellSize +
-                        voxelX;
+    const voxelOffset = this.computeVoxelOffset(x, y, z);
    cell[voxelOffset] = v;
  }
  getVoxel(x, y, z) {
    const cell = this.getCellForVoxel(x, y, z);
    if (!cell) {
      return 0;
    }
-    const {cellSize} = this;
-    const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
-    const voxelY = THREE.MathUtils.euclideanModulo(y, cellSize) | 0;
-    const voxelZ = THREE.MathUtils.euclideanModulo(z, cellSize) | 0;
-    const voxelOffset = voxelY * cellSize * cellSize +
-                        voxelZ * cellSize +
-                        voxelX;
+    const voxelOffset = this.computeVoxelOffset(x, y, z);
    return cell[voxelOffset];
  }
  generateGeometryDataForCell(cellX, cellY, cellZ) {

  ...
}
```

Filling the First Cell

Now let's make some code to fill out the first cell with voxels.

```javascript
const cellSize = 32;

const world = new VoxelWorld(cellSize);

for (let y = 0; y < cellSize; ++y) {
  for (let z = 0; z < cellSize; ++z) {
    for (let x = 0; x < cellSize; ++x) {
      const height = (Math.sin(x / cellSize * Math.PI * 2) + Math.sin(z / cellSize
      if (y < height) {
        world.setVoxel(x, y, z, 1);
      }
    }
  }
}
```

Generating BufferGeometry from Voxel Data

and some code to actually generate geometry like we covered in the article on custom BufferGeometry.

```javascript
const {positions, normals, indices} = world.generateGeometryDataForCell(0, 0, 0);
const geometry = new THREE.BufferGeometry();
const material = new THREE.MeshLambertMaterial({color: 'green'});

const positionNumComponents = 3;
const normalNumComponents = 3;
geometry.setAttribute(
    'position',
    new THREE.BufferAttribute(new Float32Array(positions), positionNumComponents));
geometry.setAttribute(
    'normal',
    new THREE.BufferAttribute(new Float32Array(normals), normalNumComponents));
geometry.setIndex(indices);
const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```

Adding Textures

let's try it

click here to open in a separate window

That seems to be working! Okay, let's add in textures.

Searching on the net I found this set of CC-BY-NC-SA licensed minecraft textures by Joshtimus. I picked a few at random and built this texture atlas.

To make things simple they are arranged a voxel type per column where the top row is the side of a voxel. The 2nd row is the top of voxel, and the 3rd row is the bottom of the voxel.

Knowing that we can add info to our `VoxelWorld.faces` data to specify for each face which row to use and the UVs to use for that face.

```javascript
```

```
VoxelWorld.faces = [
  { // left
+    uvRow: 0,
    dir: [ -1,  0,  0, ],
    corners: [
-      [ 0, 1, 0 ],
-      [ 0, 0, 0 ],
-      [ 0, 1, 1 ],
-      [ 0, 0, 1 ],
+      { pos: [ 0, 1, 0 ], uv: [ 0, 1 ], },
+      { pos: [ 0, 0, 0 ], uv: [ 0, 0 ], },
+      { pos: [ 0, 1, 1 ], uv: [ 1, 1 ], },
+      { pos: [ 0, 0, 1 ], uv: [ 1, 0 ], },
    ],
  },
  { // right
+    uvRow: 0,
    dir: [  1,  0,  0, ],
    corners: [
-      [ 1, 1, 1 ],
-      [ 1, 0, 1 ],
-      [ 1, 1, 0 ],
-      [ 1, 0, 0 ],
+      { pos: [ 1, 1, 1 ], uv: [ 0, 1 ], },
+      { pos: [ 1, 0, 1 ], uv: [ 0, 0 ], },
+      { pos: [ 1, 1, 0 ], uv: [ 1, 1 ], },
+      { pos: [ 1, 0, 0 ], uv: [ 1, 0 ], },
    ],
  },
  { // bottom
+    uvRow: 1,
    dir: [  0, -1,  0, ],
    corners: [
-      [ 1, 0, 1 ],
-      [ 0, 0, 1 ],
-      [ 1, 0, 0 ],
-      [ 0, 0, 0 ],
+      { pos: [ 1, 0, 1 ], uv: [ 1, 0 ], },
+      { pos: [ 0, 0, 1 ], uv: [ 0, 0 ], },
+      { pos: [ 1, 0, 0 ], uv: [ 1, 1 ], },
+      { pos: [ 0, 0, 0 ], uv: [ 0, 1 ], },
    ],
  },
  { // top
+    uvRow: 2,
    dir: [  0,  1,  0, ],
    corners: [
-      [ 0, 1, 1 ],
-      [ 1, 1, 1 ],
-      [ 0, 1, 0 ],
-      [ 1, 1, 0 ],
+      { pos: [ 0, 1, 1 ], uv: [ 1, 1 ], },
+      { pos: [ 1, 1, 1 ], uv: [ 0, 1 ], },
+      { pos: [ 0, 1, 0 ], uv: [ 1, 0 ], },
+      { pos: [ 1, 1, 0 ], uv: [ 0, 0 ], },
    ],
  },
  { // back
+    uvRow: 0,
    dir: [  0,  0, -1, ],
    corners: [
```

```
-       [ 1, 0, 0 ],
-       [ 0, 0, 0 ],
-       [ 1, 1, 0 ],
-       [ 0, 1, 0 ],
+       { pos: [ 1, 0, 0 ], uv: [ 0, 0 ], },
+       { pos: [ 0, 0, 0 ], uv: [ 1, 0 ], },
+       { pos: [ 1, 1, 0 ], uv: [ 0, 1 ], },
+       { pos: [ 0, 1, 0 ], uv: [ 1, 1 ], },
    ],
  },
  { // front
+    uvRow: 0,
    dir: [  0,  0,  1, ],
    corners: [
-       [ 0, 0, 1 ],
-       [ 1, 0, 1 ],
-       [ 0, 1, 1 ],
-       [ 1, 1, 1 ],
+       { pos: [ 0, 0, 1 ], uv: [ 0, 0 ], },
+       { pos: [ 1, 0, 1 ], uv: [ 1, 0 ], },
+       { pos: [ 0, 1, 1 ], uv: [ 0, 1 ], },
+       { pos: [ 1, 1, 1 ], uv: [ 1, 1 ], },
    ],
  },
];
```

Updating Geometry Generation for UVs

And we can update the code to use that data. We need to know the size of a tile in the texture atlas and the dimensions of the texture.

```
javascript
```

```
class VoxelWorld {
-  constructor(cellSize) {
-    this.cellSize = cellSize;
+  constructor(options) {
+    this.cellSize = options.cellSize;
+    this.tileSize = options.tileSize;
+    this.tileTextureWidth = options.tileTextureWidth;
+    this.tileTextureHeight = options.tileTextureHeight;
+    const {cellSize} = this;
+    this.cellSliceSize = cellSize * cellSize;
+    this.cell = new Uint8Array(cellSize * cellSize * cellSize);
  }

  ...

  generateGeometryDataForCell(cellX, cellY, cellZ) {
-    const {cellSize} = this;
+    const {cellSize, tileSize, tileTextureWidth, tileTextureHeight} = this;
    const positions = [];
    const normals = [];
+    const uvs = [];
    const indices = [];
    const startX = cellX * cellSize;
    const startY = cellY * cellSize;
    const startZ = cellZ * cellSize;

    for (let y = 0; y < cellSize; ++y) {
      const voxelY = startY + y;
      for (let z = 0; z < cellSize; ++z) {
        const voxelZ = startZ + z;
        for (let x = 0; x < cellSize; ++x) {
          const voxelX = startX + x;
          const voxel = this.getVoxel(voxelX, voxelY, voxelZ);
          if (voxel) {
            const uvVoxel = voxel - 1;  // voxel 0 is sky so for UVs we start at 0
            // There is a voxel here but do we need faces for it?
-            for (const {dir, corners} of VoxelWorld.faces) {
+            for (const {dir, corners, uvRow} of VoxelWorld.faces) {
              const neighbor = this.getVoxel(
                  voxelX + dir[0],
                  voxelY + dir[1],
                  voxelZ + dir[2]);
              if (!neighbor) {
                // this voxel has no neighbor in this direction so we need a face.
                const ndx = positions.length / 3;
-                for (const pos of corners) {
+                for (const {pos, uv} of corners) {
                  positions.push(pos[0] + x, pos[1] + y, pos[2] + z);
                  normals.push(...dir);
+                  uvs.push(
+                        (uvVoxel +   uv[0]) * tileSize / tileTextureWidth,
+                     1 - (uvRow + 1 - uv[1]) * tileSize / tileTextureHeight);
                }
                indices.push(
                  ndx, ndx + 1, ndx + 2,
                  ndx + 2, ndx + 1, ndx + 3,
                );
              }
            }
          }
        }
```

```
      }
    }

    return {
      positions,
      normals,
      uvs,
      indices,
    };
  }
}
```

Loading and Configuring the Texture

We then need to load the texture

javascript

```
const loader = new THREE.TextureLoader();
const texture = loader.load('resources/images/minecraft/flourish-cc-by-nc-sa.png',
texture.magFilter = THREE.NearestFilter;
texture.minFilter = THREE.NearestFilter;
texture.colorSpace = THREE.SRGBColorSpace;
```

Passing Settings to VoxelWorld

and pass the settings to the `VoxelWorld` class

javascript

```
+const tileSize = 16;
+const tileTextureWidth = 256;
+const tileTextureHeight = 64;
-const world = new VoxelWorld(cellSize);
+const world = new VoxelWorld({
+  cellSize,
+  tileSize,
+  tileTextureWidth,
+  tileTextureHeight,
+});
```

Using UVs and Texture in the Material

Let's actually use the UVs when we create the geometry and the texture when we make the material

javascript

```
-const {positions, normals, indices} = world.generateGeometryDataForCell(0, 0, 0);
+const {positions, normals, uvs, indices} = world.generateGeometryDataForCell(0, 0,
const geometry = new THREE.BufferGeometry();
-const material = new THREE.MeshLambertMaterial({color: 'green'});
+const material = new THREE.MeshLambertMaterial({
+  map: texture,
+  side: THREE.DoubleSide,
+  alphaTest: 0.1,
+  transparent: true,
+});

const positionNumComponents = 3;
const normalNumComponents = 3;
+const uvNumComponents = 2;
geometry.setAttribute(
    'position',
    new THREE.BufferAttribute(new Float32Array(positions), positionNumComponents));
geometry.setAttribute(
    'normal',
    new THREE.BufferAttribute(new Float32Array(normals), normalNumComponents));
+geometry.setAttribute(
+    'uv',
+    new THREE.BufferAttribute(new Float32Array(uvs), uvNumComponents));
geometry.setIndex(indices);
const mesh = new THREE.Mesh(geometry, material);
scene.add(mesh);
```

Using Multiple Voxel Types

One last thing, we actually need to set some voxels to use different textures.

javascript

```
for (let y = 0; y < cellSize; ++y) {
  for (let z = 0; z < cellSize; ++z) {
    for (let x = 0; x < cellSize; ++x) {
      const height = (Math.sin(x / cellSize * Math.PI * 2) + Math.sin(z / cellSize
      if (y < height) {
-        world.setVoxel(x, y, z, 1);
+        world.setVoxel(x, y, z, randInt(1, 17));
      }
    }
  }
}

+function randInt(min, max) {
+  return Math.floor(Math.random() * (max - min) + min);
+}
```

Result with Textures

and with that we get textures!

click here to open in a separate window

Supporting Multiple Cells

Let's make it support more than one cell.

To do this lets store cells in an object using cell ids. A cell id will just be a cell's coordinates separated by a comma. In other words if we ask for voxel 35,0,0 that is in cell 1,0,0 so its id is `"1,0,0"`.

javascript

```
class VoxelWorld {
  constructor(options) {
    this.cellSize = options.cellSize;
    this.tileSize = options.tileSize;
    this.tileTextureWidth = options.tileTextureWidth;
    this.tileTextureHeight = options.tileTextureHeight;
    const {cellSize} = this;
    this.cellSliceSize = cellSize * cellSize;
-    this.cell = new Uint8Array(cellSize * cellSize * cellSize);
+    this.cells = {};
  }
+  computeCellId(x, y, z) {
+    const {cellSize} = this;
+    const cellX = Math.floor(x / cellSize);
+    const cellY = Math.floor(y / cellSize);
+    const cellZ = Math.floor(z / cellSize);
+    return ${cellX},${cellY},${cellZ};
+  }
+  getCellForVoxel(x, y, z) {
-    const cellX = Math.floor(x / cellSize);
-    const cellY = Math.floor(y / cellSize);
-    const cellZ = Math.floor(z / cellSize);
-    if (cellX !== 0 || cellY !== 0 || cellZ !== 0) {
-      return null;
-    }
-    return this.cell;
+    return this.cells[this.computeCellId(x, y, z)];
  }

   ...
}
```

Adding New Cells Automatically

and now we can make `setVoxel` add new cells if we try to set a voxel in a cell that does not yet exist

javascript

```
setVoxel(x, y, z, v) {
-    const cell = this.getCellForVoxel(x, y, z);
+    let cell = this.getCellForVoxel(x, y, z);
    if (!cell) {
-      return 0;
+      cell = this.addCellForVoxel(x, y, z);
    }
    const voxelOffset = this.computeVoxelOffset(x, y, z);
    cell[voxelOffset] = v;
  }
+  addCellForVoxel(x, y, z) {
+    const cellId = this.computeCellId(x, y, z);
+    let cell = this.cells[cellId];
+    if (!cell) {
+      const {cellSize} = this;
+      cell = new Uint8Array(cellSize * cellSize * cellSize);
+      this.cells[cellId] = cell;
+    }
+    return cell;
+  }
```

Adding the UI

Let's make this editable.

First we`ll add a UI. Using radio buttons we can make an 8x2 array of tiles

html

```
<body>
  <canvas id="c"></canvas>
+  <div id="ui">
+    <div class="tiles">
+      <input type="radio" name="voxel" id="voxel1" value="1"><label for="voxel1" s
+      <input type="radio" name="voxel" id="voxel2" value="2"><label for="voxel2" s
+      <input type="radio" name="voxel" id="voxel3" value="3"><label for="voxel3" s
+      <input type="radio" name="voxel" id="voxel4" value="4"><label for="voxel4" s
+      <input type="radio" name="voxel" id="voxel5" value="5"><label for="voxel5" s
+      <input type="radio" name="voxel" id="voxel6" value="6"><label for="voxel6" s
+      <input type="radio" name="voxel" id="voxel7" value="7"><label for="voxel7" s
+      <input type="radio" name="voxel" id="voxel8" value="8"><label for="voxel8" s
+    </div>
+    <div class="tiles">
+      <input type="radio" name="voxel" id="voxel9"  value="9" ><label for="voxel9"
+      <input type="radio" name="voxel" id="voxel10" value="10"><label for="voxel10
+      <input type="radio" name="voxel" id="voxel11" value="11"><label for="voxel11
+      <input type="radio" name="voxel" id="voxel12" value="12"><label for="voxel12
+      <input type="radio" name="voxel" id="voxel13" value="13"><label for="voxel13
+      <input type="radio" name="voxel" id="voxel14" value="14"><label for="voxel14
+      <input type="radio" name="voxel" id="voxel15" value="15"><label for="voxel15
+      <input type="radio" name="voxel" id="voxel16" value="16"><label for="voxel16
+    </div>
+  </div>
</body>
```

CSS for the UI

And add some CSS to style it, display the tiles and highlight the current selection

```css
body {
    margin: 0;
}
#c {
    width: 100%;
    height: 100%;
    display: block;
}
+#ui {
+    position: absolute;
+    left: 10px;
+    top: 10px;
+    background: rgba(0, 0, 0, 0.8);
+    padding: 5px;
+}
+#ui input[type=radio] {
+  width: 0;
+  height: 0;
+  display: none;
+}
+#ui input[type=radio] + label {
+  background-image: url('resources/images/minecraft/flourish-cc-by-nc-sa.png');
+  background-size: 1600% 400%;
+  image-rendering: pixelated;
+  width: 64px;
+  height: 64px;
+  display: inline-block;
+}
+#ui input[type=radio]:checked + label {
+  outline: 3px solid red;
+}
+@media (max-width: 600px), (max-height: 600px) {
+  #ui input[type=radio] + label {
+    width: 32px;
+    height: 32px;
+  }
+}
```

UX Behavior

The UX will be as follows. If no tile is selected and you click a voxel that voxel will be erased or if you click a voxel and are holding the shift key it will be erased. Otherwise if a tiles is selected it will be added. You can deselect the selected tile type by clicking it again.

This code will let the user unselect the highlighted radio button.

javascript

```
let currentVoxel = 0;
let currentId;

document.querySelectorAll('#ui .tiles input[type=radio][name=voxel]').forEach((elem
  elem.addEventListener('click', allowUncheck);
});

function allowUncheck() {
  if (this.id === currentId) {
    this.checked = false;
    currentId = undefined;
    currentVoxel = 0;
  } else {
    currentId = this.id;
    currentVoxel = parseInt(this.value);
  }
}
```

Placing Voxels on Click

And this below code will let us set a voxel based on where the user clicks. It uses code similar to the code we made in the article on picking but it's not using the built in `RayCaster`. Instead it's using `VoxelWorld.intersectRay` which returns the position of intersection and the normal of the face hit.

javascript

```
function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}

function placeVoxel(event) {
  const pos = getCanvasRelativePosition(event);
  const x = (pos.x / canvas.width ) *  2 - 1;
  const y = (pos.y / canvas.height) * -2 + 1;  // note we flip Y

  const start = new THREE.Vector3();
  const end = new THREE.Vector3();
  start.setFromMatrixPosition(camera.matrixWorld);
  end.set(x, y, 1).unproject(camera);

  const intersection = world.intersectRay(start, end);
  if (intersection) {
    const voxelId = event.shiftKey ? 0 : currentVoxel;
    // the intersection point is on the face. That means
    // the math imprecision could put us on either side of the face.
    // so go half a normal into the voxel if removing (currentVoxel = 0)
    // our out of the voxel if adding (currentVoxel  > 0)
    const pos = intersection.position.map((v, ndx) => {
      return v + intersection.normal[ndx] * (voxelId > 0 ? 0.5 : -0.5);
    });
    world.setVoxel(...pos, voxelId);
    updateVoxelGeometry(...pos);
    requestRenderIfNotRequested();
  }
}

const mouse = {
  x: 0,
  y: 0,
};

function recordStartPosition(event) {
  mouse.x = event.clientX;
  mouse.y = event.clientY;
  mouse.moveX = 0;
  mouse.moveY = 0;
}
function recordMovement(event) {
  mouse.moveX += Math.abs(mouse.x - event.clientX);
  mouse.moveY += Math.abs(mouse.y - event.clientY);
}
function placeVoxelIfNoMovement(event) {
  if (mouse.moveX < 5 && mouse.moveY < 5) {
    placeVoxel(event);
  }
  window.removeEventListener('pointermove', recordMovement);
  window.removeEventListener('pointerup', placeVoxelIfNoMovement);
}
canvas.addEventListener('pointerdown', (event) => {
  event.preventDefault();
  recordStartPosition(event);
  window.addEventListener('pointermove', recordMovement);
  window.addEventListener('pointerup', placeVoxelIfNoMovement);
```

```
}, {passive: false});
canvas.addEventListener('touchstart', (event) => {
  // stop scrolling
  event.preventDefault();
}, {passive: false});
```

Mouse Behavior Explanation

There's a lot going on in the code above. Basically the mouse has a dual purpose. One is to move the camera. The other is to edit the world. Placing/Erasing a voxel happen when you let off the mouse but only if you have not moved the mouse since you first pressed down. This is just a guess that if you did move the mouse you were trying to move the camera, not place a block. `moveX` and `moveY` are in absolute movement so if you move to the left 10 and then back to the right 10 you'll have moved 20 units. In that case the user likely was just rotating the model back and forth and does not want to place a block. I didn't do any testing to see if `5` is a good range or not.

In the code we call `world.setVoxel` to set a voxel and then `updateVoxelGeometry` to update the three.js geometry based on what's changed.

Let's make that now. If the user clicks a voxel on the edge of a cell then the geometry for the voxel in the adjacent cell might need new geometry. This means we need to check the cell for the voxel we just edited as well as in all 6 directions from that cell.

javascript

```
const neighborOffsets = [
  [ 0,  0,  0], // self
  [-1,  0,  0], // left
  [ 1,  0,  0], // right
  [ 0, -1,  0], // down
  [ 0,  1,  0], // up
  [ 0,  0, -1], // back
  [ 0,  0,  1], // front
];
function updateVoxelGeometry(x, y, z) {
  const updatedCellIds = {};
  for (const offset of neighborOffsets) {
    const ox = x + offset[0];
    const oy = y + offset[1];
    const oz = z + offset[2];
    const cellId = world.computeCellId(ox, oy, oz);
    if (!updatedCellIds[cellId]) {
      updatedCellIds[cellId] = true;
      updateCellGeometry(ox, oy, oz);
    }
  }
}
```

Simplifying Adjacent Cell Updates

I thought about checking for adjacent cells like

```
const voxelX = THREE.MathUtils.euclideanModulo(x, cellSize) | 0;
if (voxelX === 0) {
  // update cell to the left
} else if (voxelX === cellSize - 1) {
  // update cell to the right
}
```

and there would be 4 more checks for the other 4 directions but it occurred to me the code would be much simpler with just an array of offsets and saving off the cell ids of the cells we already updated. If the updated voxel is not on the edge of a cell then the test will quickly reject updating the same cell.

For `updateCellGeometry` we're just going to take the code we had before that was generating the geometry for one cell and make it handle multiple cells.

javascript

```
const cellIdToMesh = {};
function updateCellGeometry(x, y, z) {
  const cellX = Math.floor(x / cellSize);
  const cellY = Math.floor(y / cellSize);
  const cellZ = Math.floor(z / cellSize);
  const cellId = world.computeCellId(x, y, z);
  let mesh = cellIdToMesh[cellId];
  const geometry = mesh ? mesh.geometry : new THREE.BufferGeometry();

  const {positions, normals, uvs, indices} = world.generateGeometryDataForCell(cell
  const positionNumComponents = 3;
  geometry.setAttribute('position', new THREE.BufferAttribute(new Float32Array(posi
  const normalNumComponents = 3;
  geometry.setAttribute('normal', new THREE.BufferAttribute(new Float32Array(normal
  const uvNumComponents = 2;
  geometry.setAttribute('uv', new THREE.BufferAttribute(new Float32Array(uvs), uvNu
  geometry.setIndex(indices);
  geometry.computeBoundingSphere();

  if (!mesh) {
    mesh = new THREE.Mesh(geometry, material);
    mesh.name = cellId;
    cellIdToMesh[cellId] = mesh;
    scene.add(mesh);
    mesh.position.set(cellX * cellSize, cellY * cellSize, cellZ * cellSize);
  }
}
```

Final Result

The code above checks a map of cell ids to meshes. If we ask for a cell that doesn't exist a new `Mesh` is made and added to the correct place in world space. At the end we update the attributes and indices with the new data.

click here to open in a separate window

Notes and Further Considerations

Some notes:

`RayCaster` might have worked just fine. I didn't try it. Instead I found a voxel specific raycaster. that is optimized for voxels.

I made `intersectRay` part of VoxelWorld because it seemed like if it gets too slow we could raycast against cells before raycasting on voxels as a simple speed up if it becomes too slow.

You might want to change the length of the raycast as currently it's all the way to Z-far. I expect if the user clicks something too far way they don't really want to be placing blocks on the other side of the world that are 1 or 2 pixel large.

Calling `geometry.computeBoundingSphere` might be slow. We could just manually set the bounding sphere to the fit the entire cell.

Do we want remove cells if all voxels in that cell are 0? That would probably be reasonable change if we wanted to ship this.

Thinking about how this works it's clear the absolute worst case is a checkerboard of on and off voxels. I don't know off the top of my head what other strategies to use if things get too slow. Maybe getting too slow would just encourage the user not to make giant checkerboard areas.

To keep it simple the texture atlas is just 1 column per voxel type. It would be better to make something more flexible where we have a table of voxel types and each type can specify where its face textures are in the atlas. As it is lots of space is wasted.

Looking at real minecraft there are tiles that are not voxels, not cubes. Like a fence tile or flowers. To do that we'd again need some table of voxel types and for each voxel whether it's a cube or some other geometry. If it's not a cube the neighbor check when generating the geometry would also need to change. A flower voxel next to another voxel should not remove the faces between them.

If you want to make some minecraft like thing using three.js I hope this has given you some ideas where to start and how to generate some what efficient geometry.

Aligning HTML Elements to 3D

GUIDE

Source: https://threejs.org/manual/en/align-html-elements-to-3d.html

This guide explains how to align HTML elements to 3D objects in a three.js scene, covering basic positioning, handling overlapping labels, frustum checks for off-screen objects, zIndex sorting for correct stacking order, and advanced techniques like dot-product visibility checks for a globe example.

Introduction

This article is part of a series of articles about three.js. The first article is three.js fundamentals. If you haven't read that yet and you're new to three.js you might want to consider starting there.

Sometimes you'd like to display some text in your 3D scene. You have many options each with pluses and minuses.

- Use 3D text

If you look at the primitives article you'll see TextGeometry which makes 3D text. This might be useful for flying logos but probably not so useful for stats, info, or labelling lots of things.

- Use a texture with 2D text drawn into it.

The article on using a Canvas as a texture shows using a canvas as a texture. You can draw text into a canvas and display it as a billboard. The plus here might be that the text is integrated into the 3D scene. For something like a computer terminal shown in a 3D scene this might be perfect.

- Use HTML Elements and position them to match the 3D

The benefits to this approach is you can use all of HTML. Your HTML can have multiple elements. It can by styled with CSS. It can also be selected by the user as it is actual text.

This article will cover this last approach.

```javascript
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
```

Basic Setup

Let's start simple. We'll make a 3D scene with a few primitives and then add a label to each primitive. We'll start with an example from the article on responsive pages.

We'll add some OrbitControls like we did in the article on lighting.

We need to provide an HTML element to contain our label elements.

By putting both the canvas and the `<div id="labels">` inside a parent container we can make them overlap with this CSS.

let's also add some CSS for the labels themselves.

Now into our code we don't have to add too much. We had a function `makeInstance` that we used to generate cubes. Let's make it so it also adds a label element.

As you can see we're adding a `<div>` to the container, one for each cube. We're also returning an object with both the `cube` and the `elem` for the label.

Calling it we need to provide a name for each.

What remains is positioning the label elements at render time.

And with that we have labels aligned to their corresponding objects.

```javascript
const controls = new OrbitControls(camera, canvas);
controls.target.set(0, 0, 0);
controls.update();
```

```html
<body>
  <div id="container">
    <canvas id="c"></canvas>
    <div id="labels"></div>
  </div>
</body>
```

css

```
#c {
    width: 100%;  /* let our container decide our size */
    height: 100%;
    display: block;
}
#container {
  position: relative;  /* makes this the origin of its children */
  width: 100%;
  height: 100%;
  overflow: hidden;
}
#labels {
  position: absolute;  /* let us position ourself inside the container */
  left: 0;             /* make our position the top left of the container */
  top: 0;
  color: white;
}
```

css

```
#labels>div {
  position: absolute;  /* let us position them inside the container */
  left: 0;             /* make their default position the top left of the container
  top: 0;
  cursor: pointer;     /* change the cursor to a hand when over us */
  font-size: large;
  user-select: none;   /* don't let the text get selected */
  text-shadow:         /* create a black outline */
    -1px -1px 0 #000,
     0   -1px 0 #000,
     1px -1px 0 #000,
     1px  0   0 #000,
     1px  1px 0 #000,
     0    1px 0 #000,
    -1px  1px 0 #000,
    -1px  0   0 #000;
}
#labels>div:hover {
  color: red;
}
```

javascript

```
const labelContainerElem = document.querySelector('#labels');

function makeInstance(geometry, color, x, name) {
  const material = new THREE.MeshPhongMaterial({color});

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  cube.position.x = x;

  const elem = document.createElement('div');
  elem.textContent = name;
  labelContainerElem.appendChild(elem);

  return {cube, elem};
}
```

javascript

```
const cubes = [
  makeInstance(geometry, 0x44aa88,  0, 'Aqua'),
  makeInstance(geometry, 0x8844aa, -2, 'Purple'),
  makeInstance(geometry, 0xaa8844,  2, 'Gold'),
];
```

javascript

```
const tempV = new THREE.Vector3();

...

cubes.forEach((cubeInfo, ndx) => {
  const {cube, elem} = cubeInfo;
  const speed = 1 + ndx * .1;
  const rot = time * speed;
  cube.rotation.x = rot;
  cube.rotation.y = rot;

  // get the position of the center of the cube
  cube.updateWorldMatrix(true, false);
  cube.getWorldPosition(tempV);

  // get the normalized screen coordinate of that position
  // x and y will be in the -1 to +1 range with x = -1 being
  // on the left and y = -1 being on the bottom
  tempV.project(camera);

  // convert the normalized position to CSS coordinates
  const x = (tempV.x *  .5 + .5) * canvas.clientWidth;
  const y = (tempV.y * -.5 + .5) * canvas.clientHeight;

  // move the elem to that position
  elem.style.transform = translate(-50%, -50%) translate(${x}px,${y}px);
});
```

Handling Overlapping Labels

There are a couple of issues we probably want to deal with.

One is that if we rotate the objects so they overlap all the labels overlap as well.

Another is that if we zoom way out so that the objects go outside the frustum the labels will still appear.

A possible solution to the problem of overlapping objects is to use the picking code from the article on picking. We'll pass in the position of the object on the screen and then ask the RayCaster to tell us which objects were intersected. If our object is not the first one then we are not in the front.

This handles overlapping.

```
javascript
```

```
const tempV = new THREE.Vector3();
const raycaster = new THREE.Raycaster();

...

cubes.forEach((cubeInfo, ndx) => {
  const {cube, elem} = cubeInfo;
  const speed = 1 + ndx * .1;
  const rot = time * speed;
  cube.rotation.x = rot;
  cube.rotation.y = rot;

  // get the position of the center of the cube
  cube.updateWorldMatrix(true, false);
  cube.getWorldPosition(tempV);

  // get the normalized screen coordinate of that position
  // x and y will be in the -1 to +1 range with x = -1 being
  // on the left and y = -1 being on the bottom
  tempV.project(camera);

  // ask the raycaster for all the objects that intersect
  // from the eye toward this object's position
  raycaster.setFromCamera(tempV, camera);
  const intersectedObjects = raycaster.intersectObjects(scene.children);
  // We're visible if the first intersection is this object.
  const show = intersectedObjects.length && cube === intersectedObjects[0].object;

  if (!show) {
    // hide the label
    elem.style.display = 'none';
  } else {
    // un-hide the label
    elem.style.display = '';

    // convert the normalized position to CSS coordinates
    const x = (tempV.x *  .5 + .5) * canvas.clientWidth;
    const y = (tempV.y * -.5 + .5) * canvas.clientHeight;

    // move the elem to that position
    elem.style.transform = translate(-50%, -50%) translate(${x}px,${y}px);
  }
});
```

Handling Labels Outside the Frustum

To handle going outside the frustum we can add this check if the origin of the object is outside the frustum by checking tempV.z.

This *kind of* works because the normalized coordinates we computed include a z value that goes from -1 when at the near part of our camera frustum to +1 when at the far part of our camera frustum.

For the frustum check, the solution above fails as we're only checking the origin of the object. For a large object. That origin might go outside the frustum but half of the

object might still be in the frustum.

A more correct solution would be to check if the object itself is in the frustum or not. Unfortunate that check is slow. For 3 cubes it will not be a problem but for many objects it might be.

Three.js provides some functions to check if an object's bounding sphere is in a frustum.

Our current overlapping solution has similar issues. Picking is slow. We could use gpu based picking like we covered in the picking article but that is also not free. Which solution you chose depends on your needs.

javascript

```
if (!show || Math.abs(tempV.z) > 1) {
    // hide the label
    elem.style.display = 'none';
```

javascript

```
// at init time
const frustum = new THREE.Frustum();
const viewProjection = new THREE.Matrix4();

...

// before checking
camera.updateMatrix();
camera.updateMatrixWorld();
camera.matrixWorldInverse.copy(camera.matrixWorld).invert();

...

// then for each mesh
someMesh.updateMatrix();
someMesh.updateMatrixWorld();

viewProjection.multiplyMatrices(
    camera.projectionMatrix, camera.matrixWorldInverse);
frustum.setFromProjectionMatrix(viewProjection);
const inFrustum = frustum.contains(someMesh));
```

Sorting Labels with zIndex

Another issue is the order the labels appear. If we change the code to have longer labels and set the CSS so these don't wrap, then we can run into an issue where the purple box is in the back but its label is in front of the aqua box.

We can fix this by setting the zIndex of each element. The projected position has a z value that goes from -1 in front to positive 1 in back. zIndex is required to be an integer and goes the opposite direction meaning for zIndex greater values are in front so the following code should work.

Because of the way the projected z value works we need to pick a large number to spread out the values otherwise many will have the same value. To make sure the labels don't overlap with other parts of the page we can tell the browser to create a new stacking context by setting the z-index of the container of the labels.

And now the labels should always be in the correct order.

javascript

```
const cubes = [
  makeInstance(geometry, 0x44aa88,  0, 'Aqua Colored Box'),
  makeInstance(geometry, 0x8844aa, -2, 'Purple Colored Box'),
  makeInstance(geometry, 0xaa8844,  2, 'Gold Colored Box'),
];
```

css

```
#labels>div {
  white-space: nowrap;
```

javascript

```
// convert the normalized position to CSS coordinates
const x = (tempV.x *  .5 + .5) * canvas.clientWidth;
const y = (tempV.y * -.5 + .5) * canvas.clientHeight;

// move the elem to that position
elem.style.transform = translate(-50%, -50%) translate(${x}px,${y}px);

// set the zIndex for sorting
elem.style.zIndex = (-tempV.z * .5 + .5) * 100000 | 0;
```

css

```
#labels {
  position: absolute;  /* let us position ourself inside the container */
  z-index: 0;          /* make a new stacking context so children don't sort with r
  left: 0;             /* make our position the top left of the container */
  top: 0;
  color: white;
  z-index: 0;
}
```

Globe Example - Loading Country Data

While we're at it let's do one more example to show one more issue. Let's draw a globe like Google Maps and label the countries.

I found this data which contains the borders of countries. It's licensed as CC-BY-SA.

I wrote some code to load the data, and generate country outlines and some JSON data with the names of the countries and their locations.

The JSON data is an array of entries something like this where min, max, lat, lon, are all in latitude and longitude degrees.

Let's load it up. The code is based on the examples from optimizing lots of objects though we are not drawing lots of objects we'll be using the same solutions for rendering on demand.

The first thing is to make a sphere and use the outline texture.

Then let's load the JSON file by first making a loader and then calling it.

```json
[
  {
    "name": "Algeria",
    "min": [
      -8.667223,
      18.976387
    ],
    "max": [
      11.986475,
      37.091385
    ],
    "area": 238174,
    "lat": 28.163,
    "lon": 2.632,
    "population": {
      "2005": 32854159
    }
  },
  ...
```

```javascript
```

```
{
  const loader = new THREE.TextureLoader();
  const texture = loader.load('resources/data/world/country-outlines-4k.png', rende
  const geometry = new THREE.SphereGeometry(1, 64, 32);
  const material = new THREE.MeshBasicMaterial({map: texture});
  scene.add(new THREE.Mesh(geometry, material));
}
```

javascript

```
async function loadJSON(url) {
  const req = await fetch(url);
  return req.json();
}
```

javascript

```
let countryInfos;
async function loadCountryData() {
  countryInfos = await loadJSON('resources/data/world/country-info.json');
    ...
  }
  requestRenderIfNotRequested();
}
loadCountryData();
```

Globe Example - Placing Labels

Now let's use that data to generate and place the labels.

In the article on optimizing lots of objects we had setup a small scene graph of helper objects to make it easy to compute latitude and longitude positions on our globe. See that article for an explanation of how they work.

We'll use that to compute a position for each label.

The code above looks very similar to the code we wrote for making cube labels making an element per label. When we're done we have an array, countryInfos, with one entry for each country to which we've added an elem property for the label element for that country and a position with its position on the globe.

Just like we did for the cubes we need to update the position of the labels and render time.

You can see the code above is substantially similar to the cube example before. The only major difference is we pre-computed the label positions at init time. We can do

this because the globe never moves. Only our camera moves.

Lastly we need to call updateLabels in our render loop.

And this is what we get. That is way too many labels!

We have 2 problems:

- Labels facing away from us are showing up.
- There are too many labels.

javascript

```
const lonFudge = Math.PI * 1.5;
const latFudge = Math.PI;
// these helpers will make it easy to position the boxes
// We can rotate the lon helper on its Y axis to the longitude
const lonHelper = new THREE.Object3D();
// We rotate the latHelper on its X axis to the latitude
const latHelper = new THREE.Object3D();
lonHelper.add(latHelper);
// The position helper moves the object to the edge of the sphere
const positionHelper = new THREE.Object3D();
positionHelper.position.z = 1;
latHelper.add(positionHelper);
```

javascript

```
const labelParentElem = document.querySelector('#labels');
for (const countryInfo of countryInfos) {
  const {lat, lon, name} = countryInfo;

  // adjust the helpers to point to the latitude and longitude
  lonHelper.rotation.y = THREE.MathUtils.degToRad(lon) + lonFudge;
  latHelper.rotation.x = THREE.MathUtils.degToRad(lat) + latFudge;

  // get the position of the lat/lon
  positionHelper.updateWorldMatrix(true, false);
  const position = new THREE.Vector3();
  positionHelper.getWorldPosition(position);
  countryInfo.position = position;

  // add an element for each country
  const elem = document.createElement('div');
  elem.textContent = name;
  labelParentElem.appendChild(elem);
  countryInfo.elem = elem;
```

javascript

```
const tempV = new THREE.Vector3();

function updateLabels() {
  // exit if we have not yet loaded the JSON file
  if (!countryInfos) {
    return;
  }

  for (const countryInfo of countryInfos) {
    const {position, elem} = countryInfo;

    // get the normalized screen coordinate of that position
    // x and y will be in the -1 to +1 range with x = -1 being
    // on the left and y = -1 being on the bottom
    tempV.copy(position);
    tempV.project(camera);

    // convert the normalized position to CSS coordinates
    const x = (tempV.x *  .5 + .5) * canvas.clientWidth;
    const y = (tempV.y * -.5 + .5) * canvas.clientHeight;

    // move the elem to that position
    elem.style.transform = translate(-50%, -50%) translate(${x}px,${y}px);

    // set the zIndex for sorting
    elem.style.zIndex = (-tempV.z * .5 + .5) * 100000 | 0;
  }
}
```

javascript

```
function render() {
  renderRequested = false;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

  controls.update();

  updateLabels();

  renderer.render(scene, camera);
}
```

Hiding Back-Facing Labels

For issue #1 we can't really use the RayCaster like we did above as there is nothing to intersect except the sphere. Instead what we can do is check if that particular country is facing away from us or not. This works because the label positions are around a sphere. In fact we're using a unit sphere, a sphere with a radius of 1.0. That means the positions are already unit directions making the math relatively easy.

Above we use the positions as a direction and get that direction relative to the camera. Then we get the camera relative direction from the camera to that position on the globe and take the dot product. The dot product returns the cosine of the angle between the to vectors. This gives us a value from -1 to +1 where -1 means the label is facing the camera, 0 means the label is directly on the edge of the sphere relative to the camera, and anything greater than zero is behind. We then use that value to show or hide the element.

In the diagram we can see the dot product of the direction the label is facing to direction from the camera to that position. If you rotate the direction you'll see the dot product is -1.0 when the direction is directly facing the camera, it's 0.0 when exactly on the tangent of the sphere relative to the camera or to put it another way it's 0 when the 2 vectors are perpendicular to each other, 90 degrees It's greater than zero with the label is behind the sphere.

```javascript
```

```
const tempV = new THREE.Vector3();
const cameraToPoint = new THREE.Vector3();
const cameraPosition = new THREE.Vector3();
const normalMatrix = new THREE.Matrix3();

function updateLabels() {
  // exit if we have not yet loaded the JSON file
  if (!countryInfos) {
    return;
  }

  const minVisibleDot = 0.2;
  // get a matrix that represents a relative orientation of the camera
  normalMatrix.getNormalMatrix(camera.matrixWorldInverse);
  // get the camera's position
  camera.getWorldPosition(cameraPosition);
  for (const countryInfo of countryInfos) {
    const {position, elem} = countryInfo;

    // Orient the position based on the camera's orientation.
    // Since the sphere is at the origin and the sphere is a unit sphere
    // this gives us a camera relative direction vector for the position.
    tempV.copy(position);
    tempV.applyMatrix3(normalMatrix);

    // compute the direction to this position from the camera
    cameraToPoint.copy(position);
    cameraToPoint.applyMatrix4(camera.matrixWorldInverse).normalize();

    // get the dot product of camera relative direction to this position
    // on the globe with the direction from the camera to that point.
    // 1 = facing directly towards the camera
    // 0 = exactly on tangent of the sphere from the camera
    // < 0 = facing away
    const dot = tempV.dot(cameraToPoint);

    // if the orientation is not facing us hide it.
    if (dot < minVisibleDot) {
      elem.style.display = 'none';
      continue;
    }

    // restore the element to its default display style
    elem.style.display = '';

    // get the normalized screen coordinate of that position
    // x and y will be in the -1 to +1 range with x = -1 being
    // on the left and y = -1 being on the bottom
    tempV.copy(position);
    tempV.project(camera);

    // convert the normalized position to CSS coordinates
    const x = (tempV.x *  .5 + .5) * canvas.clientWidth;
    const y = (tempV.y * -.5 + .5) * canvas.clientHeight;

    // move the elem to that position
    countryInfo.elem.style.transform = translate(-50%, -50%) translate(${x}px,${y}p

    // set the zIndex for sorting
    elem.style.zIndex = (-tempV.z * .5 + .5) * 100000 | 0;
```

```
  }
}
```

Filtering Labels by Area and Adding GUI

For issue #2, too many labels we need some way to decide which labels to show. One way would be to only show labels for large countries. The data we're loading contains min and max values for the area a country covers. From that we can compute an area and then use that area to decide whether or not to display the country.

At init time let's compute the area.

Then at render time let's use the area to decide to display the label or not.

Finally, since I'm not sure what good values are for these settings lets add a GUI so we can play with them.

You can see as you rotate the earth labels that go behind disappear. Adjust the minVisibleDot to see the cutoff change. You can also adjust the minArea value to see larger or smaller countries appear.

The more I worked on this the more I realized just how much work is put into Google Maps. They have also have to decide which labels to show. I'm pretty sure they use all kinds of criteria. For example your current location, your default language setting, your account settings if you have an account, they probably use population or popularity, they might give priority to the countries in the center of the view, etc ... Lots to think about.

In any case I hope these examples gave you some idea of how to align HTML elements with your 3D.

```javascript
```

```
const labelParentElem = document.querySelector('#labels');
for (const countryInfo of countryInfos) {
  const {lat, lon, min, max, name} = countryInfo;

  // adjust the helpers to point to the latitude and longitude
  lonHelper.rotation.y = THREE.MathUtils.degToRad(lon) + lonFudge;
  latHelper.rotation.x = THREE.MathUtils.degToRad(lat) + latFudge;

  // get the position of the lat/lon
  positionHelper.updateWorldMatrix(true, false);
  const position = new THREE.Vector3();
  positionHelper.getWorldPosition(position);
  countryInfo.position = position;

  // compute the area for each country
  const width = max[0] - min[0];
  const height = max[1] - min[1];
  const area = width * height;
  countryInfo.area = area;

  // add an element for each country
  const elem = document.createElement('div');
  elem.textContent = name;
  labelParentElem.appendChild(elem);
  countryInfo.elem = elem;
}
```

javascript

```
const large = 20 * 20;
const maxVisibleDot = 0.2;
// get a matrix that represents a relative orientation of the camera
normalMatrix.getNormalMatrix(camera.matrixWorldInverse);
// get the camera's position
camera.getWorldPosition(cameraPosition);
for (const countryInfo of countryInfos) {
  const {position, elem, area} = countryInfo;
  // large enough?
  if (area < large) {
    elem.style.display = 'none';
    continue;
  }

  ...
```

javascript

```
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
import {GUI} from 'three/addons/libs/lil-gui.module.min.js';
```

javascript

```
const settings = {
  minArea: 20,
  maxVisibleDot: -0.2,
};
const gui = new GUI({width: 300});
gui.add(settings, 'minArea', 0, 50).onChange(requestRenderIfNotRequested);
gui.add(settings, 'maxVisibleDot', -1, 1, 0.01).onChange(requestRenderIfNotRequeste

function updateLabels() {
  if (!countryInfos) {
    return;
  }

  const large = settings.minArea * settings.minArea;
  // get a matrix that represents a relative orientation of the camera
  normalMatrix.getNormalMatrix(camera.matrixWorldInverse);
  // get the camera's position
  camera.getWorldPosition(cameraPosition);
  for (const countryInfo of countryInfos) {

    ...

    // if the orientation is not facing us hide it.
    if (dot > settings.maxVisibleDot) {
      elem.style.display = 'none';
      continue;
    }
```

WebGPU

WebGPURenderer

GUIDE

Source: https://threejs.org/manual/en/webgpurenderer.html

An overview of three.js's next-generation WebGPURenderer, which uses WebGPU by default with a WebGL 2 fallback, including usage instructions, code examples, and migration considerations from WebGLRenderer.

WebGPURenderer

The new `WebGPURenderer` is the next-generation renderer for three.js. This article provides a short overview about the new renderer and basic guidelines about the usage.

Overview

`WebGPURenderer` is designed to be the modern alternative to the long-standing `WebGLRenderer`. Its primary goal is to use WebGPU, which is a modern, high-performance graphics and compute 3D API. However, it's built to be a universal renderer. If a device/browser doesn't support WebGPU, the renderer can automatically fall back to using a WebGL 2 backend.

Providing a WebGL 2 backend as a fallback is a crucial design decision since it allows applications to benefit from WebGPU but without sacrificing the support for devices which only support WebGL 2.

Apart from the fact that `WebGPURenderer` enables access to WebGPU, it offers an exciting feature set:

- `WebGPURenderer` comes with a new node-based material system which allows to develop custom materials with greater flexibility and more robustness.
- The renderer supports TSL, the three.js shading language. With TSL, developers can write shader code with JavaScript in a platform-independent manner. Shader code written in TSL can be transpiled to WGSL or GLSL depending on the available backend.
- `WebGPURenderer` comes with a new post-processing stack with built-in Multiple Render Targets (MRT) support and automatic pass combination thanks to the node material.

Let's find out how to integrate `WebGPURenderer` in three.js applications.

Usage

`WebGPURenderer` has different build files so the way you import three.js changes:

If you are using an import map, it's recommended to change it to the following (the paths differ depending on your setup):

You can create an instance of the renderer just like with `WebGLRenderer`:

It's important to understand that WebGPU is initialized in an asynchronous fashion. Hence, it is recommended to use `setAnimationLoop()` to define the animation loop of your app since this approach will automatically ensure the renderer is initialized when rendering the first frame. If you prefer to manage your animation loop

via `window.requestAnimationFrame()` or if you have to use the renderer in your init routine, you need an additional line in the above code section.

Most common methods known from `WebGLRenderer` like `clear()`, `setRenderTarget()` or `dispose()` are also present in `WebGPURenderer`. Please have a look at the API documentation for a full overview of the renderer's public interface.

Like mentioned in the initial part of the guide, `WebGPURenderer` uses a WebGPU backend by default and a WebGL 2 backend as a fallback. If you want to force the usage of WebGL 2 for testing purposes or if you want to exclude the usage of WebGPU for certain reasons, you can make use of the `forceWebGL` parameter.

javascript

```
-  import * as THREE from 'three';
+  import * as THREE from 'three/webgpu';
```

html

```
<script type="importmap">
  {
    "imports": {
      "three": "../build/three.webgpu.js",
      "three/webgpu": "../build/three.webgpu.js",
      "three/tsl": "../build/three.tsl.js",
      "three/addons/": "./jsm/"
    }
  }
</script>
```

javascript

```
const renderer = new THREE.WebGPURenderer( { antialias: true } );
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight );
renderer.setAnimationLoop( render );
document.body.appendChild( renderer.domElement );
```

javascript

```
const renderer = new THREE.WebGPURenderer( { antialias: true } );
renderer.setPixelRatio( window.devicePixelRatio );
renderer.setSize( window.innerWidth, window.innerHeight );
renderer.setAnimationLoop( render );
document.body.appendChild( renderer.domElement );

+  await renderer.init();
```

```javascript
- const renderer = new THREE.WebGPURenderer( { antialias: true } );
+ const renderer = new THREE.WebGPURenderer( { antialias: true, forceWebGL: true }
```

Migration

If you want to give `WebGPURenderer` a try, you have to be aware of the following.

- Custom materials based on `ShaderMaterial`, `RawShaderMaterial` and modifications of built-in materials via `onBeforeCompile()` are not supported in `WebGPURenderer`. This part of your application must be ported to node materials and TSL.
- `EffectComposer` with its effect passes are not supported because `WebGPURenderer` comes with a new, more modern post-processing stack. Similar to materials, post-processing effects are now written in TSL and the effect chain is expressed as a node composition. All common effects have already been ported to `WebGPURenderer` and exist in a more performant version as a node class. We have also added new effects like SSGI, SSS or a better DoF exclusively for the new renderer. Check out the official examples to get an overview of the current supported effects.
- The renderer itself is still in an experimental state although its maturity level has been greatly improved in the last years. Still, depending on your application and scene setup, you will encounter missing features or a better performance with `WebGLRenderer`. Feel free to file an issue at GitHub so we are aware of open tasks. We are improving `WebGPURenderer` with each release so it's recommended to upgrade to the latest version whenever possible.

State of WebGLRenderer

Although in the meanwhile a lot of work happens in context of `WebGPURenderer`, the node material and TSL, `WebGLRenderer` is still maintained and the recommended choice for pure WebGL 2 applications. However, keep in mind that there are no plans to add larger new features to the renderer since the project's focus is now on `WebGPURenderer` which you can easily see at the latest release notes. That said, we are currently investigating the possibility to add limited node material support to `WebGLRenderer` in order to make the transition to `WebGPURenderer` easier for certain projects.

Post-Processing with WebGPURenderer

GUIDE

Source: https://threejs.org/manual/en/webgpu-postprocessing.html

This guide explains how to use the new post-processing stack built into WebGPURenderer, including MRT support, effect composition via TSL nodes, tone mapping control, and advanced MRT packing techniques.

Overview

The previous post-processing for `WebGLRenderer` had many conceptual issues. Making use of Multiple Render Targets (MRT) was cumbersome due to the limited support in the renderer and there was no automatic pass/effect combination to improve the overall performance.

The new post-processing stack for `WebGPURenderer` was designed to support these use cases right from the beginning.

- `WebGPURenderer` comes with full, built-in MRT support.
- The system combines effects if possible which reduces the overall number of render passes.
- The effect chain is expressed as a node composition which allows a more flexible effect setup.

Let's find out how to integrate post-processing in three.js applications.

Basics

First, please read the instructions in the guide about WebGPURenderer to correctly configure your imports. After that, you can create an instance of the render pipleine module like so:

The instance of `RenderPipeline` replaces the previous instance of `EffectComposer`. To make sure you actually use the output of the module, you have to update your animation loop like so:

Many post-processing setups start with a so called "scene pass" or "beauty pass" that represents the image of you rendered scene. This image should be subsequently enhanced by different effects like Bloom, Depth-of-Field or SSR. Start by importing the `pass()` TSL function from the TSL namespace and use it to create the pass.

The basic idea of the node system is to represent materials or post-processing effects as node compositions. To configure a basic Dotscreen and RGB shift effect, you create effect nodes with TSL functions and compose them together.

When you are done, you can simply assign the final node to the `RenderPipeline` instance.

javascript

```javascript
const renderPipeline = new THREE.RenderPipeline( renderer );
```

diff

```diff
-  renderer.render( scene, camera );
+  renderPipeline.render();
```

javascript

```javascript
import { pass } from 'three/tsl';

// in your init routine

const scenePass = pass( scene, camera );
```

diff

```diff
import { pass } from 'three/tsl';
+  import { dotScreen } from 'three/addons/tsl/display/DotScreenNode.js';
+  import { rgbShift } from 'three/addons/tsl/display/RGBShiftNode.js';

// in your init routine

const scenePass = pass( scene, camera );

+  const dotScreenPass = dotScreen( scenePass );
+  const rgbShiftPass = rgbShift( dotScreenPass );
```

javascript

```javascript
renderPipeline.outputNode = rgbShiftPass;
```

Tone Mapping and Color Spaces

When using post-processing, tone mapping and color space conversion are automatically applied at the end of your effect chain. Sometimes you want full control

over how and when these steps are executed though. For example if you want to apply FXAA with `FXAANode` or color grading with `Lut3DNode`, you can disable automatic tone mapping and color space conversion and apply it via `renderOutput()` by yourself.

It is not mandatory to use `renderOutput()`, you can also implement a custom tone mapping and color space conversion based on your requirements.

javascript

```
import { pass, renderOutput } from 'three/tsl';
import { fxaa } from 'three/addons/tsl/display/FXAANode.js';

// in your init routine

const renderPipeline = new THREE.RenderPipeline( renderer );
renderPipeline.outputColorTransform = false; // disable default output color transf

const scenePass = pass( scene, camera );
const outputPass = renderOutput( scenePass ); // apply tone mapping and color space

// FXAA must be computed in sRGB color space

const fxaaPass = fxaa( outputPass );
renderPipeline.outputNode = fxaaPass;
```

MRT

The new post-processing stack has built-in Multiple Render Targets (MRT) support which is crucial for more advanced setups. MRT allows you to produce multiple outputs in a single render pass. So for example when rendering your scene with TRAA, you need below setup to prepare the inputs for the anti-aliasing.

The configuration object you assign to the `mrt()` TSL function describes the different outputs of the pass. In this case, we save the default output (the scene's beauty) and scene's velocity since we want to setup a TRAA. If you also require the scene's depth, there is no need to configure it as a MRT output. You get it for free in your default output pass if you request it in your app. If you know want to use these outputs in subsequent effects, you can query them as texture nodes.

The MRT configuration varies depending on your setup. There are many different TSL objects like `output`, `velocity`, `normalView` or `emissive` than you can use to save per-fragment data in MRT attachments. To improve performance and avoid hitting memory restrictions, it's important to pack and optimize your data in complex MRT setups. By default all attachments are RGBA16 (Half-Float) in precision which is not necessary for all types of data. As an example, below code queries the `diffuseColor`

attachment and sets its format to RGBA8 which cuts down the memory and bandwidth by half.

Below setup for Scree-Space Reflections (SSR) converts the default FP16 normals into RGBA8 colors and packs metalness/roughness into a single attachment. The usage of the `sample()` TSL functions allows to implement custom unpacking. In this instance, it converts the color back to a (normalized) direction vector.

We want to further improve the packing/unpacking features in the future to offer more ways to pack/unpack MRT data. In the meanwhile, please have a look at the official examples to get an overview about the existing effects and setups.

javascript

```
import { pass, mrt, output, velocity } from 'three/tsl';

// in your init routine

const scenePass = pass( scene, camera );
scenePass.setMRT( mrt( {
  output: output,
  velocity: velocity
} ) );
```

javascript

```
import { traa } from 'three/addons/tsl/display/TRAANode.js';

// in your init routine

const scenePassColor = scenePass.getTextureNode( 'output' );
const scenePassDepth = scenePass.getTextureNode( 'depth' );
const scenePassVelocity = scenePass.getTextureNode( 'velocity' );

const traaPass = traa( scenePassColor, scenePassDepth, scenePassVelocity, camera );
renderPipeline.outputNode = traaPass;
```

javascript

```
const diffuseTexture = scenePass.getTexture( 'diffuseColor' );
diffuseTexture.type = THREE.UnsignedByteType;
```

javascript

```
scenePass.setMRT( mrt( {
  output: output,
  normal: packNormalToRGB( normalView ),
  metalrough: vec2( metalness, roughness )
} ) );

// use RGBA8 instead of RGBA16

const normalTexture = scenePass.getTexture( 'normal' );
normalTexture.type = THREE.UnsignedByteType;

const metalRoughTexture = scenePass.getTexture( 'metalrough' );
metalRoughTexture.type = THREE.UnsignedByteType;

// custom unpacking. use the resulting "sceneNormal" instead of "scenePassNormal"
// in subsequent effects

const sceneNormal = sample( ( uv ) => {

  return unpackRGBToNormal( scenePassNormal.sample( uv ) );

} );
```

Virtual Reality

How to create VR content

GUIDE

Source: https://threejs.org/manual/en/how-to-create-vr-content.html

This guide provides a brief overview of the basic components of a web-based VR application made with three.js, covering VRButton setup, XR rendering enablement, and the animation loop.

Overview

This guide provides a brief overview of the basic components of a web-based VR application made with three.js.

Workflow

First, you have to include VRButton.js into your project.

VRButton.createButton() does two important things: It creates a button which indicates VR compatibility. Besides, it initiates a VR session if the user activates the button. The only thing you have to do is to add the following line of code to your app.

Next, you have to tell your instance of WebGLRenderer to enable XR rendering.

Finally, you have to adjust your animation loop since we can't use our well known window.requestAnimationFrame() function. For VR projects we use renderer.setAnimationLoop(). The minimal code looks like this:

javascript

```javascript
import { VRButton } from 'three/addons/webxr/VRButton.js';
```

javascript

```javascript
document.body.appendChild( VRButton.createButton( renderer ) );
```

javascript

```javascript
renderer.xr.enabled = true;
```

javascript

```javascript
renderer.setAnimationLoop( function () {

  renderer.render( scene, camera );

} );
```

Next Steps

Have a look at one of the official WebVR examples to see this workflow in action.

WebXR / XR / ballshooter WebXR / XR / cubes WebXR / XR / dragging WebXR / XR / marching cubes WebXR / XR / paint WebXR / VR / panorama_depth WebXR / VR / panorama WebXR / VR / rollercoaster WebXR / VR / sandbox WebXR / VR / video

VR

GUIDE

Source: https://threejs.org/manual/en/webxr-basics.html

A guide to building VR applications with three.js using WebXR, covering setup, device considerations, and supporting both VR and non-VR modes.

Introduction

Making a VR app in three.js is pretty simple. You basically just have to tell three.js you want to use WebXR. If you think about it a few things about WebXR should be clear. Which way the camera is pointing is supplied by the VR system itself since the user turns their head to choose a direction to look. Similarly the field of view and aspect will be supplied by the VR system since each system has a different field of view and display aspect.

Let's take an example from the article on making a responsive webpage and make it support VR.

Requirements

Before we get started you're going to need a VR capable device like an Android smartphone, Google Daydream, Oculus Go, Oculus Rift, Vive, Samsung Gear VR., an iPhone with a WebXR browser.

Next, if you are running locally you need to run a simple web server like is covered in the article on setting up.

If the device you are using to view VR is not the same computer you're running on you need to serve your webpage via https or else the browser will not allow using the WebXR API. The server mentioned in the article on setting up called Servez has an option to use https. Check it and start the server.

The note the URLs. You need the one that is your computer's local ipaddress. It will usually start with `192`, `172` or `10`. Type that full address, including the `https://` part into your VR device's browser. Note: Your computer and your VR device need to be on the same local network or WiFi and you probably need to be on a home network. note: Many cafes are setup to disallow this kind of machine to machine connection.

You'll be greeted with an error something like the one below. Click "advanced" and then click *proceed*.

Now you can run your examples.

If you're really going to do WebXR development another thing you should learn about is remote debugging so that you can see console warnings, errors, and of course actually debug your code.

If you just want to see the code work below you can just run the code from this site.

Basic Setup

The first thing we need to do is include the VR support after including three.js.

Then we need to enable three.js's WebXR support and add its VR button to our page.

```javascript
import * as THREE from 'three';
import {VRButton} from 'three/addons/webxr/VRButton.js';
```

```javascript
function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
  renderer.xr.enabled = true;
  document.body.appendChild(VRButton.createButton(renderer));
```

Render Loop

We need to let three.js run our render loop. Until now we have used a `requestAnimationFrame` loop but to support VR we need to let three.js handle our render loop for us. We can do that by calling `WebGLRenderer.setAnimationLoop` and passing a function to call for the loop.

javascript

```
function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

  cubes.forEach((cube, ndx) => {
    const speed = 1 + ndx * .1;
    const rot = time * speed;
    cube.rotation.x = rot;
    cube.rotation.y = rot;
  });

  renderer.render(scene, camera);

  requestAnimationFrame(render);
}

requestAnimationFrame(render);
renderer.setAnimationLoop(render);
```

Camera Positioning and Units in VR

There is one more detail. We should probably set a camera height that's kind of average for a standing user.

and move the cubes up to be in front of the camera

We set them to `z = -2` since the camera will now be at `z = 0` and camera defaults to looking down the -z axis.

This brings up an extremely important point. **Units in VR are in meters**. In other words **One Unit = One Meter**. This means the camera is 1.6 meters above 0. The cube's centers are 2 meters in front of the camera. Each cube is 1x1x1 meter large. This is important because VR needs to adjust things to the user *in the real world*. That means we need the units used in three.js to match the user's own movements.

And with that we should get 3 spinning cubes in front of the camera with a button to enter VR.

click here to open in a separate window

javascript

```
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
camera.position.set(0, 1.6, 0);
```

javascript

```
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);

cube.position.x = x;
cube.position.y = 1.6;
cube.position.z = -2;
```

Adding a Background

I find that VR works better if we have something surrounding the camera like room for reference so let's add a simple grid cubemap like we covered in the article on backgrounds. We'll just use the same grid texture for each side of the cube which will give as a grid room.

That's better.

click here to open in a separate window

Note: To actually see VR you will need a WebXR compatible device. I believe most Android Phones can support WebXR using Chrome or Firefox. For iOS you might be able to use this WebXR App though in general WebXR support on iOS is unsupported as of May 2019.

javascript

```
const scene = new THREE.Scene();
{
  const loader = new THREE.CubeTextureLoader();
  const texture = loader.load([
    'resources/images/grid-1024.png',
    'resources/images/grid-1024.png',
    'resources/images/grid-1024.png',
    'resources/images/grid-1024.png',
    'resources/images/grid-1024.png',
    'resources/images/grid-1024.png',
  ]);
  scene.background = texture;
}
```

VR Device Considerations

To use WebXR on Android or iPhone you'll need a *VR Headset* for phones. You can get them for anywhere from $5 for one made of cardboard to $100. Unfortunately I don't know which ones to recommend. I've purchased 6 of them over the years and they are all of varying quality. I've never paid more than about $25.

Just to mention some of the issues

- **Do they fit your phone**

Phones come in a variety of sizes and so the VR headsets need to match. Many headsets claim to match a large variety of sizes. My experience is the more sizes they match the worse they actually are since instead of being designed for a specific size they have to make compromises to match more sizes. Unfortunately multi-size headsets are the most common type.

- **Can they focus for your face**

Some devices have more adjustments than others. Generally there are at most 2 adjustments. How far the lenses are from your eyes and how far apart the lenses are.

- **Are they too reflective**

Many headsets of a cone of plastic from your eye to the phone. If that plastic is shinny or reflective then it will act like a mirror reflecting the screen and be very distracting.

Few if any of the reviews seem to cover this issue.

- **Are the comfortable on your face.**

Most of the devices rest on your nose like a pair of glasses. That can hurt after a few minutes. Some have straps that go around your head. Others have a 3rd strap that goes over your head. These may or may not help keep the device at the right place.

It turns out for most (all?) devices, you eyes need to be centered with the lenses. If the lenses are slightly above or below your eyes the image gets out of focus. This can be very frustrating as things might start in focus but 45-60 seconds later the device has shifted up or down 1 millimeter and you suddenly realize you've been struggling to focus on a blurry image.

- **Can they support your glasses.**

If you wear eye glasses then you'll need to read the reviews to see if a particular headset works well with eye glasses.

I really can't make any recommendations unfortunately. Google has some cheap recommendations made from cardboard some of them as low as $5 so maybe start there and if you enjoy it then consider upgrading. $5 is like the price of 1 coffee so seriously, give it try!

Types of VR Devices

There are also 3 basic types of devices.

- **3 degrees of freedom (3dof), no input device**

This is generally the phone style although sometimes you can buy a 3rd party input device. The 3 degrees of freedom mean you can look up/down (1), left/right(2) and you can tilt your head left and right (3).

- **3 degrees of freedom (3dof) with 1 input device (3dof)**

This is basically Google Daydream and Oculus GO

These also allow 3 degrees of freedom and include a small controller that acts like a laser pointer inside VR. The laser pointer also only has 3 degrees of freedom. The system can tell which way the input device is pointing but it can not tell where the device is.

- **6 degrees of freedom (6dof) with input devices (6dof)**

These are *the real deal* haha. 6 degrees of freedom means not only do these device know which way you are looking but they also know where your head actually is. That means if you move from left to right or forward and back or stand up / sit down the devices can register this and everything in VR moves accordingly. It's spookily and amazingly real feeling. With a good demo you'll be blown away or at least I was and still am.

Further these devices usually include 2 controllers, one for each hand and the system can tell exactly where your hands are and which way they are oriented and so you can manipulate things in VR by just reaching out, touching, pushing, twisting, etc...

6 degree of freedom devices include the Vive and Vive Pro, the Oculus Rift and Quest, and I believe all of the Windows MR devices.

With all that covered I don't for sure know which devices will work with WebXR. I'm 99% sure that most Android phones will work when running Chrome. You may need to turn on WebXR support in `about:flags`. I also know Google Daydream will also work and similarly you need to enable WebXR support in `about:flags`. Oculus Rift, Vive, and Vive Pro will work via Chrome or Firefox. I'm less sure about Oculus Go and Oculus Quest as both of them use custom OSes but according to the internet they both appear to work.

Supporting both VR and Non-VR

Okay, after that long detour about VR Devices and WebXR there's some things to cover

AFAICT, at least as of r112, there is no easy way to support both VR and non-VR modes with three.js. Ideally if not in VR mode you'd be able to control the camera using whatever means you want, for example the `OrbitControls`, and you'd get some kind of event when switching into and out of VR mode so that you could turn the controls on/off.

If three.js adds some support to do both I'll try to update this article. Until then you might need 2 versions of your site OR pass in a flag in the URL, something like

Then we could add some links in to switch modes

and some CSS to position them

in your code you could use that parameter like this

Whether that's good or bad I don't know. I have a feeling the differences between what's needed for VR and what's needed for non-VR are often very different so for all but the most simple things maybe 2 separate pages are better? You'll have to decide.

Note for various reasons this will not work in the live editor on this site so if you want to check it out click here. It should start in non-VR mode and you can use the mouse or fingers to move the camera. Clicking "Allow VR" should switch to allow VR mode and you should be able to click "Enter VR" if you're on a VR device.

```text
https://mysite.com/mycooldemo?allowvr=true
```

```html
```

```
<body>
  <canvas id="c"></canvas>
  <div class="mode">
    <a href="?allowvr=true" id="vr">Allow VR</a>
    <a href="?" id="nonvr">Use Non-VR Mode</a>
  </div>
</body>
```

css

```
body {
    margin: 0;
}
#c {
    width: 100%;
    height: 100%;
    display: block;
}
.mode {
  position: absolute;
  right: 1em;
  top: 1em;
}
```

javascript

```
function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
  renderer.xr.enabled = true;
  document.body.appendChild(VRButton.createButton(renderer));

  const fov = 75;
  const aspect = 2;  // the canvas default
  const near = 0.1;
  const far = 5;
  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
  camera.position.set(0, 1.6, 0);

  const params = (new URL(document.location)).searchParams;
  const allowvr = params.get('allowvr') === 'true';
  if (allowvr) {
    renderer.xr.enabled = true;
    document.body.appendChild(VRButton.createButton(renderer));
    document.querySelector('#vr').style.display = 'none';
  } else {
    // no VR, add some controls
    const controls = new OrbitControls(camera, canvas);
    controls.target.set(0, 1.6, -2);
    controls.update();
    document.querySelector('#nonvr').style.display = 'none';
  }
```

Deciding on the Level of VR Support

Above we covered 3 types of VR devices.

- 3DOF no input
- 3DOF + 3DOF input
- 6DOF + 6DOF input

You need to decide how much effort you're willing to put in to support each type of device.

For example the simplest device has no input. The best you can generally do is make it so there are some buttons or objects in the user's view and if the user aligns some marker in the center of the display on those objects for 1/2 a second or so then that button is clicked. A common UX is to display a small timer that will appear over the object indicating if you keep the marker there for a moment the object/button will be selected.

Since there is no other input that's about the best you can do

The next level up you have one 3DOF input device. Generally it can point at things and the user has at least 2 buttons. The Daydream also has a touchpad which provides normal touch inputs.

In any case if a user has this type of device it's far more comfortable for the user to by able to point at things with their controller than it is to make them do it with their head by looking at things.

A similar level to that might be 3DOF or 6DOF device with a game console controller. You'll have to decide what to do here. I suspect the most common thing is the user still has to look to point and the controller is just used for buttons.

The last level is a user with a 6DOF headset and 2 6DOF controllers. Those users will find an experience that is only 3DOF to often be frustrating. Similarly they usually expect to be able to virtually manipulate things with their hands in VR so you'll have to decide if you want to support that or not.

As you can see getting started in VR is pretty easy but actually making something shippable in VR will require lots of decision making and design.

This was a pretty brief intro into VR with three.js. We'll cover some of the input methods in future articles.

VR - Look to Select

GUIDE

Source: https://threejs.org/manual/en/webxr-look-to-select.html

A tutorial on implementing a 'look to select' interaction pattern for Google Cardboard-style VR in Three.js, where users select objects by pointing their head at them. Covers building a PickHelper, creating a visual selection gauge using texture offset animation, and pairing objects for selection feedback.

Introduction

NOTE: The examples on this page require a VR capable device. Without one they won't work.

In the previous article we went over a very simple VR example using three.js and we discussed the various kinds of VR systems.

The simplest and possibly most common is the Google Cardboard style of VR which is basically a phone put into a $5 - $50 face mask. This kind of VR has no controller so people have to come up with creative solutions for allowing user input.

The most common solution is "look to select" where if the user points their head at something for a moment it gets selected.

Let's implement "look to select"! We'll start with an example from the previous article and to do it we'll add the `PickHelper` we made in the article on picking.

Adding PickHelper for Basic Object Picking

For an explanation of that code see the article on picking.

To use it we just need to create an instance and call it in our render loop.

In the original picking example we converted the mouse coordinates from CSS pixels into normalized coordinates that go from -1 to +1 across the canvas.

In this case though we will always pick where the camera is facing which is the center of the screen so we pass in `0` for both `x` and `y` which is the center in normalized coordinates.

And with that objects will flash when we look at them.

javascript

```
class PickHelper {
  constructor() {
    this.raycaster = new THREE.Raycaster();
    this.pickedObject = null;
    this.pickedObjectSavedColor = 0;
  }
  pick(normalizedPosition, scene, camera, time) {
    // restore the color if there is a picked object
    if (this.pickedObject) {
      this.pickedObject.material.emissive.setHex(this.pickedObjectSavedColor);
      this.pickedObject = undefined;
    }

    // cast a ray through the frustum
    this.raycaster.setFromCamera(normalizedPosition, camera);
    // get the list of objects the ray intersected
    const intersectedObjects = this.raycaster.intersectObjects(scene.children);
    if (intersectedObjects.length) {
      // pick the first object. It's the closest one
      this.pickedObject = intersectedObjects[0].object;
      // save its color
      this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
      // set its emissive color to flashing red/yellow
      this.pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFFFF00 : 0x
    }
  }
}
```

javascript

```
+const pickHelper = new PickHelper();

...
function render(time) {
  time *= 0.001;

  ...

+  // 0, 0 is the center of the view in normalized coordinates.
+  pickHelper.pick({x: 0, y: 0}, scene, camera, time);
```

Creating a Selection Timer Gauge

Typically we don't want selection to be immediate. Instead we require the user to keep the camera on the thing they want to select for a few moments to give them a chance not to select something by accident.

To do that we need some kind of meter or gauge or some way to convey that the user must keep looking and for how long.

One easy way we could do that is to make a 2 color texture and use a texture offset to slide the texture across a model.

Let's do this by itself to see it work before we add it to the VR example.

First we make an `OrthographicCamera`.

And of course update it if the canvas changes size.

We now have a camera that shows 2 units above and below the center and aspect units left and right.

Next let's make a 2 color texture. We'll use a `DataTexture` which we've used a few other places.

We'll then use that texture on a `TorusGeometry`.

`THREE.MathUtils.mapLinear` takes a value that goes between `fromStart` and `fromEnd` and maps it to a value between `toStart` and `toEnd`. In the case above we're taking `time % 2` which means a value that goes from 0 to 2 and maps that to a value that goes from -0.5 to 0.5.

Textures are mapped to geometry using normalized texture coordinates that go from 0 to 1. That means our 2x1 pixel image, set to the default wrapping mode of `THREE.ClampToEdge`, if we adjust the texture coordinates by -0.5 then the entire mesh will be the first color and if we adjust the texture coordinates by +0.5 the entire mesh will be the second color. In between with the filtering set to `THREE.NearestFilter` we'll be able to move the transition between the 2 colors through the geometry.

javascript

```javascript
const left = -2;    // Use values for left
const right = 2;    // right, top and bottom
const top = 1;      // that match the default
const bottom = -1;  // canvas size.
const near = -1;
const far = 1;
const camera = new THREE.OrthographicCamera(left, right, top, bottom, near, far);
```

javascript

```
function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    const aspect = canvas.clientWidth / canvas.clientHeight;
+    camera.left = -aspect;
+    camera.right = aspect;
    camera.updateProjectionMatrix();
  }
  ...
```

javascript

```
function makeDataTexture(data, width, height) {
  const texture = new THREE.DataTexture(data, width, height, THREE.RGBAFormat);
  texture.minFilter = THREE.NearestFilter;
  texture.magFilter = THREE.NearestFilter;
  texture.needsUpdate = true;
  return texture;
}

const cursorColors = new Uint8Array([
  64, 64, 64, 64,       // dark gray
  255, 255, 255, 255,   // white
]);
const cursorTexture = makeDataTexture(cursorColors, 2, 1);
```

javascript

```
const ringRadius = 0.4;
const tubeRadius = 0.1;
const tubeSegments = 4;
const ringSegments = 64;
const cursorGeometry = new THREE.TorusGeometry(
    ringRadius, tubeRadius, tubeSegments, ringSegments);

const cursorMaterial = new THREE.MeshBasicMaterial({
  color: 'white',
  map: cursorTexture,
  transparent: true,
  blending: THREE.CustomBlending,
  blendSrc: THREE.OneMinusDstColorFactor,
  blendDst: THREE.OneMinusSrcColorFactor,
});
const cursor = new THREE.Mesh(cursorGeometry, cursorMaterial);
scene.add(cursor);
```

javascript

```
function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    const aspect = canvas.clientWidth / canvas.clientHeight;
    camera.left = -aspect;
    camera.right = aspect;
    camera.updateProjectionMatrix();
  }

+  const fromStart = 0;
+  const fromEnd = 2;
+  const toStart = -0.5;
+  const toEnd = 0.5;
+  cursorTexture.offset.x = THREE.MathUtils.mapLinear(
+      time % 2,
+      fromStart, fromEnd,
+      toStart, toEnd);

  renderer.render(scene, camera);
}
```

javascript

```
+const backgroundColors = new Uint8Array([
+    0,   0,   0, 255,  // black
+   90,  38,  38, 255,  // dark red
+  100, 175, 103, 255,  // medium green
+  255, 239, 151, 255,  // light yellow
+]);
+const backgroundTexture = makeDataTexture(backgroundColors, 2, 2);
+backgroundTexture.wrapS = THREE.RepeatWrapping;
+backgroundTexture.wrapT = THREE.RepeatWrapping;
+backgroundTexture.repeat.set(4, 4);

const scene = new THREE.Scene();
+scene.background = backgroundTexture;
```

Notes on Cursor Implementation

A few things to notice **and try**.

- We set the `cursorMaterial`'s `blending`, `blendSrc` and `blendDst` properties as follows:

```
    blending: THREE.CustomBlending,
    blendSrc: THREE.OneMinusDstColorFactor,
    blendDst: THREE.OneMinusSrcColorFactor,
```

This gives as an *inverse* type of effect. Comment out those 3 lines and you'll see the difference. I'm just guessing the inverse effect is best here as that way we can hopefully see the cursor regardless of the colors it is over.

- We use a `TorusGeometry` and not a `RingGeometry`.

  For whatever reason the `RingGeometry` uses a flat UV mapping scheme. Because of this if we use a `RingGeometry` the texture slides horizontally across the ring instead of around it like it does above.

  Try it out, change the `TorusGeometry` to a `RingGeometry` (it's just commented out in the example above) and you'll see what I mean.

  The *proper* thing to do (for some definition of *proper*) would be to either use the `RingGeometry` but fix the texture coordinates so they go around the ring. Or else, generate our own ring geometry. But, the torus works just fine. Placed directly in front of the camera with a `MeshBasicMaterial` it will look exactly like a ring and the texture coordinates go around the ring so it works for our needs.

Integrating the Cursor into VR Code

Let's integrate it with our VR code above.

You can see the code above we added all the code to create the cursor geometry, texture, and material and we added it as a child of the camera so it will always be in front of the camera. Note we need to add the camera to the scene otherwise the cursor won't be rendered.

We then check if the thing we're picking this time is the same as it was last time. If so we add the elapsed time to a timer and if the timer reaches its limit we return the selected item.

```
javascript
```

```
class PickHelper {
-  constructor() {
+  constructor(camera) {
    this.raycaster = new THREE.Raycaster();
    this.pickedObject = null;
-    this.pickedObjectSavedColor = 0;

+    const cursorColors = new Uint8Array([
+      64, 64, 64, 64,       // dark gray
+      255, 255, 255, 255,   // white
+    ]);
+    this.cursorTexture = makeDataTexture(cursorColors, 2, 1);
+
+    const ringRadius = 0.4;
+    const tubeRadius = 0.1;
+    const tubeSegments = 4;
+    const ringSegments = 64;
+    const cursorGeometry = new THREE.TorusGeometry(
+        ringRadius, tubeRadius, tubeSegments, ringSegments);
+
+    const cursorMaterial = new THREE.MeshBasicMaterial({
+      color: 'white',
+      map: this.cursorTexture,
+      transparent: true,
+      blending: THREE.CustomBlending,
+      blendSrc: THREE.OneMinusDstColorFactor,
+      blendDst: THREE.OneMinusSrcColorFactor,
+    });
+    const cursor = new THREE.Mesh(cursorGeometry, cursorMaterial);
+    // add the cursor as a child of the camera
+    camera.add(cursor);
+    // and move it in front of the camera
+    cursor.position.z = -1;
+    const scale = 0.05;
+    cursor.scale.set(scale, scale, scale);
+    this.cursor = cursor;
+
+    this.selectTimer = 0;
+    this.selectDuration = 2;
+    this.lastTime = 0;
  }
  pick(normalizedPosition, scene, camera, time) {
+    const elapsedTime = time - this.lastTime;
+    this.lastTime = time;

-    // restore the color if there is a picked object
-    if (this.pickedObject) {
-      this.pickedObject.material.emissive.setHex(this.pickedObjectSavedColor);
-      this.pickedObject = undefined;
-    }

+    const lastPickedObject = this.pickedObject;
+    this.pickedObject = undefined;

    // cast a ray through the frustum
    this.raycaster.setFromCamera(normalizedPosition, camera);
    // get the list of objects the ray intersected
    const intersectedObjects = this.raycaster.intersectObjects(scene.children);
    if (intersectedObjects.length) {
      // pick the first object. It's the closest one
      this.pickedObject = intersectedObjects[0].object;
```

```
-      // save its color
-      this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
-      // set its emissive color to flashing red/yellow
-      this.pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFFFF00 : 0
    }

+    // show the cursor only if it's hitting something
+    this.cursor.visible = this.pickedObject ? true : false;
+
+    let selected = false;
+
+    // if we're looking at the same object as before
+    // increment time select timer
+    if (this.pickedObject && lastPickedObject === this.pickedObject) {
+      this.selectTimer += elapsedTime;
+      if (this.selectTimer >= this.selectDuration) {
+        this.selectTimer = 0;
+        selected = true;
+      }
+    } else {
+      this.selectTimer = 0;
+    }
+
+    // set cursor material to show the timer state
+    const fromStart = 0;
+    const fromEnd = this.selectDuration;
+    const toStart = -0.5;
+    const toEnd = 0.5;
+    this.cursorTexture.offset.x = THREE.MathUtils.mapLinear(
+        this.selectTimer,
+        fromStart, fromEnd,
+        toStart, toEnd);
+
+    return selected ? this.pickedObject : undefined;
  }
}
```

javascript

```
+scene.add(camera);
```

Creating Paired Box and Sphere Meshes

Now let's use that to pick the cubes. As a simple example we'll add 3 spheres as well. When a cube is picked with hide the cube and un-hide the corresponding sphere.

So first we'll make a sphere geometry.

Then let's create 3 pairs of box and sphere meshes. We'll use a `Map` so that we can associate each `Mesh` with its partner.

In `render` where we rotate the cubes we need to iterate over `meshToMeshMap` instead of `cubes`.

javascript

```
const boxWidth = 1;
const boxHeight = 1;
const boxDepth = 1;
-const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
+const boxGeometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
+
+const sphereRadius = 0.5;
+const sphereGeometry = new THREE.SphereGeometry(sphereRadius);
```

javascript

```
-const cubes = [
-  makeInstance(geometry, 0x44aa88,  0),
-  makeInstance(geometry, 0x8844aa, -2),
-  makeInstance(geometry, 0xaa8844,  2),
-];
+const meshToMeshMap = new Map();
+[
+  { x:  0, boxColor: 0x44aa88, sphereColor: 0xFF4444, },
+  { x:  2, boxColor: 0x8844aa, sphereColor: 0x44FF44, },
+  { x: -2, boxColor: 0xaa8844, sphereColor: 0x4444FF, },
+].forEach((info) => {
+  const {x, boxColor, sphereColor} = info;
+  const sphere = makeInstance(sphereGeometry, sphereColor, x);
+  const box = makeInstance(boxGeometry, boxColor, x);
+  // hide the sphere
+  sphere.visible = false;
+  // map the sphere to the box
+  meshToMeshMap.set(box, sphere);
+  // map the box to the sphere
+  meshToMeshMap.set(sphere, box);
+});
```

javascript

```
-cubes.forEach((cube, ndx) => {
+let ndx = 0;
+for (const mesh of meshToMeshMap.keys()) {
  const speed = 1 + ndx * .1;
  const rot = time * speed;
-  cube.rotation.x = rot;
-  cube.rotation.y = rot;
-});
+  mesh.rotation.x = rot;
+  mesh.rotation.y = rot;
+  ++ndx;
+}
```

Final Selection Logic

And now we can use our new `PickHelper` implementation to select one of the objects. When selected we hide that object and un-hide its partner.

And with that we should have a pretty decent *look to select* implementation.

```
javascript
```

```
// 0, 0 is the center of the view in normalized coordinates.
-pickHelper.pick({x: 0, y: 0}, scene, camera, time);
+const selectedObject = pickHelper.pick({x: 0, y: 0}, scene, camera, time);
+if (selectedObject) {
+  selectedObject.visible = false;
+  const partnerObject = meshToMeshMap.get(selectedObject);
+  partnerObject.visible = true;
+}
```

Conclusion

I hope this example gave some ideas of how to implement a "look to select" type of Google Cardboard level UX. Sliding textures using texture coordinates offsets is also a commonly useful technique.

Next up let's allow the user that has a VR controller to point at and move things.

VR - 3DOF Point to Select

GUIDE

Source: https://threejs.org/manual/en/webxr-point-to-select.html

A tutorial on implementing point-to-select functionality using VR controllers in Three.js, covering raycasting from controller positions, event dispatching, highlighting, and moving objects via controller interaction.

VR - 3DOF Point to Select

NOTE: The examples on this page require a VR capable device with a pointing device. Without one they won't work. See this article as to why

In the previous article we went over a very simple VR example where we let the user choose things by pointing via looking. In this article we will take it one step further and let the user choose with a pointing device

Three.js makes is relatively easy by providing 2 controller objects in VR and tries to handle both cases of a single 3DOF controller and two 6DOF controllers. Each of the

controllers are Object3D objects which give the orientation and position of that controller. They also provide `selectstart`, `select` and `selectend` events when the user starts pressing, is pressing, and stops pressing (ends) the "main" button on the controller.

Starting with the last example from the previous article let's change the `PickHelper` into a `ControllerPickHelper`.

Our new implementation will emit a `select` event that gives us the object that was picked so to use it we'll just need to do this.

Remember from our previous code `meshToMeshMap` maps our boxes and spheres to each other so if we have one we can look up its partner through `meshToMeshMap` so here we're just hiding the selected object and un-hiding its partner.

As for the actual implementation of `ControllerPickHelper`, first we need to add the VR controller objects to the scene and to those add some 3D lines we can use to display where the user is pointing. We save off both the controllers and their lines.

Without doing anything else this alone would give us 1 or 2 lines in the scene showing where the user's pointing devices are and which way they are pointing.

One problem we have though, we don't want have our `RayCaster` pick the line itself so an easy solution is separate the objects we wanted to be able to pick from the objects we don't by parenting them under another Object3D.

Next let's add some code to pick from the controllers. This is the first time we've picked with something not the camera. In our article on picking the user uses the mouse or finger to pick which means picking comes from the camera into the screen. In the previous article we were picking based on which way the user is looking so again that comes from the camera. This time though we're picking from the position of the controllers so we're not using the camera.

Like before we use a Raycaster but this time we take the ray from the controller. Our previous `PickHelper` there was only one thing picking but here we have up to 2 controllers, one for each hand. We save off which object each controller is looking at in `controllerToObjectMap`. We also save off the original emissive color in `objectToColorMap` and we make the line long enough to touch whatever it's pointing at.

We need to add some code to reset these settings every frame.

Next we want to emit a `select` event when the user clicks the controller. To do that we can extend three.js's EventDispatcher and then we'll check when we get a `select` event from the controller, then if that controller is pointing at something we emit what that controller is pointing at as our own `select` event.

All that is left is to call `update` in our render loop

and assuming you have a VR device with a controller you should be able to use the controllers to pick things.

And what if we wanted to be able to move the objects?

That's relatively easy. Let's move our controller 'select' listener code out into a function so we can use it for more than one thing.

Then let's use it for both `selectstart` and `select`

and let's also pass on the `selectend` event which three.js sends out when you user lets of the button on the controller.

Now let's change the code so when we get a `selectstart` event we'll remove the selected object from the scene and make it a child of the controller. This means it will move with the controller. When we get a `selectend` event we'll put it back in the scene.

When an object is selected we save off that object and its original parent. When the user is done we can put the object back.

We use the Object3D.attach to re-parent the selected objects. These functions let us change the parent of an object without changing its orientation and position in the scene.

And with that we should be able to move the objects around with a 6DOF controller or at least change their orientation with a 3DOF controller

To be honest I'm not 100% sure this `ControllerPickHelper` is the best way to organize the code but it's useful to demonstrating the various parts of getting something simple working in VR in three.js

```javascript
```

```
const pickHelper = new ControllerPickHelper(scene);
pickHelper.addEventListener('select', (event) => {
  event.selectedObject.visible = false;
  const partnerObject = meshToMeshMap.get(event.selectedObject);
  partnerObject.visible = true;
});
```

javascript

```
class ControllerPickHelper {
  constructor(scene) {
    const pointerGeometry = new THREE.BufferGeometry().setFromPoints([
      new THREE.Vector3(0, 0, 0),
      new THREE.Vector3(0, 0, -1),
    ]);

    this.controllers = [];
    for (let i = 0; i < 2; ++i) {
      const controller = renderer.xr.getController(i);
      scene.add(controller);

      const line = new THREE.Line(pointerGeometry);
      line.scale.z = 5;
      controller.add(line);
      this.controllers.push({controller, line});
    }
  }
}
```

javascript

```
const scene = new THREE.Scene();
+// object to put pickable objects on so we can easily
+// separate them from non-pickable objects
+const pickRoot = new THREE.Object3D();
+scene.add(pickRoot);

...

function makeInstance(geometry, color, x) {
  const material = new THREE.MeshPhongMaterial({color});

  const cube = new THREE.Mesh(geometry, material);
-  scene.add(cube);
+  pickRoot.add(cube);

...
```

javascript

```
class ControllerPickHelper {
  constructor(scene) {
+    this.raycaster = new THREE.Raycaster();
+    this.objectToColorMap = new Map();
+    this.controllerToObjectMap = new Map();
+    this.tempMatrix = new THREE.Matrix4();

    const pointerGeometry = new THREE.BufferGeometry().setFromPoints([
      new THREE.Vector3(0, 0, 0),
      new THREE.Vector3(0, 0, -1),
    ]);

    this.controllers = [];
    for (let i = 0; i < 2; ++i) {
      const controller = renderer.xr.getController(i);
      scene.add(controller);

      const line = new THREE.Line(pointerGeometry);
      line.scale.z = 5;
      controller.add(line);
      this.controllers.push({controller, line});
    }
  }
+  update(pickablesParent, time) {
+    this.reset();
+    for (const {controller, line} of this.controllers) {
+      // cast a ray through the from the controller
+      this.tempMatrix.identity().extractRotation(controller.matrixWorld);
+      this.raycaster.ray.origin.setFromMatrixPosition(controller.matrixWorld);
+      this.raycaster.ray.direction.set(0, 0, -1).applyMatrix4(this.tempMatrix);
+      // get the list of objects the ray intersected
+      const intersections = this.raycaster.intersectObjects(pickablesParent.childr
+      if (intersections.length) {
+        const intersection = intersections[0];
+        // make the line touch the object
+        line.scale.z = intersection.distance;
+        // pick the first object. It's the closest one
+        const pickedObject = intersection.object;
+        // save which object this controller picked
+        this.controllerToObjectMap.set(controller, pickedObject);
+        // highlight the object if we haven't already
+        if (this.objectToColorMap.get(pickedObject) === undefined) {
+          // save its color
+          this.objectToColorMap.set(pickedObject, pickedObject.material.emissive.g
+          // set its emissive color to flashing red/yellow
+          pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFF2000 : 0x
+        }
+      } else {
+        line.scale.z = 5;
+      }
+    }
+  }
}
```

javascript

```
class ControllerPickHelper {

  ...

+  _reset() {
+    // restore the colors
+    this.objectToColorMap.forEach((color, object) => {
+      object.material.emissive.setHex(color);
+    });
+    this.objectToColorMap.clear();
+    this.controllerToObjectMap.clear();
+  }
  update(pickablesParent, time) {
+    this._reset();

    ...

}
```

javascript

```
-class ControllerPickHelper {
+class ControllerPickHelper extends THREE.EventDispatcher {
  constructor(scene) {
+    super();
    this.raycaster = new THREE.Raycaster();
    this.objectToColorMap = new Map();  // object to save color and picked object
    this.controllerToObjectMap = new Map();
    this.tempMatrix = new THREE.Matrix4();

    const pointerGeometry = new THREE.BufferGeometry().setFromPoints([
      new THREE.Vector3(0, 0, 0),
      new THREE.Vector3(0, 0, -1),
    ]);

    this.controllers = [];
    for (let i = 0; i < 2; ++i) {
      const controller = renderer.xr.getController(i);
+      controller.addEventListener('select', (event) => {
+        const controller = event.target;
+        const selectedObject = this.controllerToObjectMap.get(controller);
+        if (selectedObject) {
+          this.dispatchEvent({type: 'select', controller, selectedObject});
+        }
+      });
      scene.add(controller);

      const line = new THREE.Line(pointerGeometry);
      line.scale.z = 5;
      controller.add(line);
      this.controllers.push({controller, line});
    }
  }
}
```

javascript

```
function render(time) {

  ...

+  pickHelper.update(pickablesParent, time);

  renderer.render(scene, camera);
}
```

javascript

```
class ControllerPickHelper extends THREE.EventDispatcher {
  constructor(scene) {
    super();

    ...

    this.controllers = [];

+    const selectListener = (event) => {
+      const controller = event.target;
+      const selectedObject = this.controllerToObjectMap.get(event.target);
+      if (selectedObject) {
+        this.dispatchEvent({type: 'select', controller, selectedObject});
+      }
+    };

    for (let i = 0; i < 2; ++i) {
      const controller = renderer.xr.getController(i);
-      controller.addEventListener('select', (event) => {
-        const controller = event.target;
-        const selectedObject = this.controllerToObjectMap.get(event.target);
-        if (selectedObject) {
-          this.dispatchEvent({type: 'select', controller, selectedObject});
-        }
-      });
+      controller.addEventListener('select', selectListener);

       ...
```

javascript

```
class ControllerPickHelper extends THREE.EventDispatcher {
  constructor(scene) {
    super();

    ...

    this.controllers = [];

    const selectListener = (event) => {
      const controller = event.target;
      const selectedObject = this.controllerToObjectMap.get(event.target);
      if (selectedObject) {
-        this.dispatchEvent({type: 'select', controller, selectedObject});
+        this.dispatchEvent({type: event.type, controller, selectedObject});
      }
    };

    for (let i = 0; i < 2; ++i) {
      const controller = renderer.xr.getController(i);
      controller.addEventListener('select', selectListener);
      controller.addEventListener('selectstart', selectListener);

       ...
```

javascript

```
class ControllerPickHelper extends THREE.EventDispatcher {
  constructor(scene) {
    super();

    ...

    this.controllers = [];

    const selectListener = (event) => {
      const controller = event.target;
      const selectedObject = this.controllerToObjectMap.get(event.target);
      if (selectedObject) {
        this.dispatchEvent({type: event.type, controller, selectedObject});
      }
    };

+    const endListener = (event) => {
+      const controller = event.target;
+      this.dispatchEvent({type: event.type, controller});
+    };

    for (let i = 0; i < 2; ++i) {
      const controller = renderer.xr.getController(i);
      controller.addEventListener('select', selectListener);
      controller.addEventListener('selectstart', selectListener);
+      controller.addEventListener('selectend', endListener);

       ...
```

javascript

```
const pickHelper = new ControllerPickHelper(scene);
-pickHelper.addEventListener('select', (event) => {
-  event.selectedObject.visible = false;
-  const partnerObject = meshToMeshMap.get(event.selectedObject);
-  partnerObject.visible = true;
-});

+const controllerToSelection = new Map();
+pickHelper.addEventListener('selectstart', (event) => {
+  const {controller, selectedObject} = event;
+  const existingSelection = controllerToSelection.get(controller);
+  if (!existingSelection) {
+    controllerToSelection.set(controller, {
+      object: selectedObject,
+      parent: selectedObject.parent,
+    });
+    controller.attach(selectedObject);
+  }
+});
+
+pickHelper.addEventListener('selectend', (event) => {
+  const {controller} = event;
+  const selection = controllerToSelection.get(controller);
+  if (selection) {
+    controllerToSelection.delete(controller);
+    selection.parent.attach(selection.object);
+  }
+});
```

Optimization & Performance

Rendering on Demand

GUIDE

Source: https://threejs.org/manual/en/rendering-on-demand.html

A guide explaining how to render Three.js scenes on demand rather than continuously using requestAnimationFrame, to save power and battery on devices where continuous animation is unnecessary.

Rendering on Demand

The topic might be obvious to many people but just in case ... most Three.js examples render continuously. In other words they setup a `requestAnimationFrame` loop or " *rAF loop* " something like this

For something that animates this makes sense but what about for something that does not animate? In that case rendering continuously is a waste of the devices power and if the user is on portable device it wastes the user's battery.

The most obvious way to solve this is to render once at the start and then render only when something changes. Changes include textures or models finally loading, data arriving from some external source, the user adjusting a setting or the camera or giving other relevant input.

Let's take an example from the article on responsiveness and modify it to render on demand.

First we'll add in the `OrbitControls` so there is something that could change that we can render in response to.

and set them up

Since we won't be animating the cubes anymore we no longer need to keep track of them

We can remove the code to animate the cubes and the calls to `requestAnimationFrame`

then we need to render once

We need to render anytime the `OrbitControls` change the camera settings. Fortunately the `OrbitControls` dispatch a `change` event anytime something changes.

We also need to handle the case where the user resizes the window. That was handled automatically before since we were rendering continuously but now what we are not we need to render when the window changes size.

And with that we get something that renders on demand.

click here to open in a separate window

The `OrbitControls` have options to add a kind of inertia to make them feel less stiff. We can enable this by setting the `enableDamping` property to true.

With `enableDamping` on we need to call `controls.update` in our render function so that the `OrbitControls` can continue to give us new camera settings as they smooth out the movement. But, that means we can't call `render` directly from the `change` event because we'll end up in an infinite loop. The controls will send us a `change` event and call `render`, `render` will call `controls.update`. `controls.update` will send another `change` event.

We can fix that by using `requestAnimationFrame` to call `render` but we need to make sure we only ask for a new frame if one has not already been requested which we can do by keeping a variable that tracks if we've already requested a frame.

We should probably also use `requestRenderIfNotRequested` for resizing as well

It might be hard to see the difference. Try clicking on the example below and use the arrow keys to move around or dragging to spin. Then try clicking on the example above and do the same thing and you should be able to tell the difference. The one above snaps when you press an arrow key or drag, the one below slides.

click here to open in a separate window

Let's also add a simple lil-gui GUI and make its changes render on demand.

Let's allow setting the color and x scale of each cube. To be able to set the color we'll use the `ColorGUIHelper` we created in the article on lights.

First we need to create a GUI

and then for each cube we'll create a folder and add 2 controls, one for `material.color` and another for `cube.scale.x`.

You can see above lil-gui controls have an `onChange` method that you can pass a callback to be called when the GUI changes a value. In our case we just need it to call `requestRenderIfNotRequested`. The call to `folder.open` makes the folder start expanded.

click here to open in a separate window

I hope this gives some idea of how to make three.js render on demand instead of continuously. Apps/pages that render three.js on demand are not as common as most pages using three.js are either games or 3d animated art but examples of pages that might be better rendering on demand would be say a map viewer, a 3d editor, a 3d graph generator, a product catalog, etc...

javascript

```
function render() {
  ...
  requestAnimationFrame(render);
}
requestAnimationFrame(render);
```

javascript

```
import * as THREE from 'three';
+import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
```

javascript

```
const fov = 75;
const aspect = 2;  // the canvas default
const near = 0.1;
const far = 5;
const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
camera.position.z = 2;

+const controls = new OrbitControls(camera, canvas);
+controls.target.set(0, 0, 0);
+controls.update();
```

javascript

```
-const cubes = [
-  makeInstance(geometry, 0x44aa88,  0),
-  makeInstance(geometry, 0x8844aa, -2),
-  makeInstance(geometry, 0xaa8844,  2),
-];
+makeInstance(geometry, 0x44aa88,  0);
+makeInstance(geometry, 0x8844aa, -2);
+makeInstance(geometry, 0xaa8844,  2);
```

javascript

```
-function render(time) {
-  time *= 0.001;
+function render() {

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

-  cubes.forEach((cube, ndx) => {
-    const speed = 1 + ndx * .1;
-    const rot = time * speed;
-    cube.rotation.x = rot;
-    cube.rotation.y = rot;
-  });

  renderer.render(scene, camera);

-  requestAnimationFrame(render);
}

-requestAnimationFrame(render);
```

javascript

```
render();
```

javascript

```
controls.addEventListener('change', render);
```

javascript

```
window.addEventListener('resize', render);
```

javascript

```
controls.enableDamping = true;
```

javascript

```
+let renderRequested = false;

function render() {
+  renderRequested = false;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

  renderer.render(scene, camera);
}
render();

+function requestRenderIfNotRequested() {
+  if (!renderRequested) {
+    renderRequested = true;
+    requestAnimationFrame(render);
+  }
+}

-controls.addEventListener('change', render);
+controls.addEventListener('change', requestRenderIfNotRequested);
```

javascript

```
-window.addEventListener('resize', render);
+window.addEventListener('resize', requestRenderIfNotRequested);
```

javascript

```
import * as THREE from 'three';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
+import {GUI} from 'three/addons/libs/lil-gui.module.min.js';
```

javascript

```
const gui = new GUI();
```

javascript

```
function makeInstance(geometry, color, x) {
  const material = new THREE.MeshPhongMaterial({color});

  const cube = new THREE.Mesh(geometry, material);
  scene.add(cube);

  cube.position.x = x;

+  const folder = gui.addFolder(Cube${x});
+  folder.addColor(new ColorGUIHelper(material, 'color'), 'value')
+      .name('color')
+      .onChange(requestRenderIfNotRequested);
+  folder.add(cube.scale, 'x', .1, 1.5)
+      .name('scale x')
+      .onChange(requestRenderIfNotRequested);
+  folder.open();

  return cube;
}
```

Tips

GUIDE

Source: https://threejs.org/manual/en/tips.html

A collection of small tips and solutions for common issues encountered when using three.js, including taking screenshots, keyboard input, transparency, and background animations.

Taking A Screenshot of the Canvas

In the browser there are effectively 2 functions that will take a screenshot. The old one `canvas.toDataURL` and the new better one `canvas.toBlob`

So you'd think it would be easy to take a screenshot by just adding some code like

click here to open in a separate window

When I tried it I got this screenshot

Yes, it's just a black image.

It's possible it worked for you depending on your browser/OS but in general it's not likely to work.

The issue is that for performance and compatibility reasons, by default the browser will clear a WebGL canvas's drawing buffer after you've drawn to it.

The solution is to call your rendering code just before capturing.

In our code we need to adjust a few things. First let's separate out the rendering code.

Now that `render` is only concerned with actually rendering we can call it just before capturing the canvas.

And now it should work.

click here to open in a separate window

For a different solution see the next item.

html

```
<canvas id="c"></canvas>
+<button id="screenshot" type="button">Save...</button>
```

javascript

```
const elem = document.querySelector('#screenshot');
elem.addEventListener('click', () => {
  canvas.toBlob((blob) => {
    saveBlob(blob, `screencapture-${canvas.width}x${canvas.height}.png`);
  });
});

const saveBlob = (function() {
  const a = document.createElement('a');
  document.body.appendChild(a);
  a.style.display = 'none';
  return function saveData(blob, fileName) {
     const url = window.URL.createObjectURL(blob);
     a.href = url;
     a.download = fileName;
     a.click();
  };
}());
```

javascript

```
+const state = {
+  time: 0,
+};

-function render(time) {
-  time *= 0.001;
+function render() {
  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

  cubes.forEach((cube, ndx) => {
    const speed = 1 + ndx * .1;
-    const rot = time * speed;
+    const rot = state.time * speed;
    cube.rotation.x = rot;
    cube.rotation.y = rot;
  });

  renderer.render(scene, camera);

-  requestAnimationFrame(render);
}

+function animate(time) {
+  state.time = time * 0.001;
+
+  render();
+
+  requestAnimationFrame(animate);
+}
+requestAnimationFrame(animate);
```

javascript

```
const elem = document.querySelector('#screenshot');
elem.addEventListener('click', () => {
+  render();
  canvas.toBlob((blob) => {
    saveBlob(blob, `screencapture-${canvas.width}x${canvas.height}.png`);
  });
});
```

Preventing the canvas being cleared

Let's say you wanted to let the user paint with an animated object. You need to pass in `preserveDrawingBuffer: true` when you create the `WebGLRenderer`. This prevents the browser from clearing the canvas. You also need to tell three.js not to clear the canvas as well.

click here to open in a separate window

Note that if you were serious about making a drawing program this would not be a solution as the browser will still clear the canvas anytime we change its resolution. We're changing is resolution based on its display size. Its display size changes when the window changes size. That includes when the user downloads a file, even in another tab, and the browser adds a status bar. It also includes when the user turns their phone and the browser switches from portrait to landscape.

If you really wanted to make a drawing program you'd render to a texture using a render target.

javascript

```
const canvas = document.querySelector('#c');
-const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
+const renderer = new THREE.WebGLRenderer({
+  canvas,
+  preserveDrawingBuffer: true,
+  alpha: true,
+});
+renderer.autoClearColor = false;
```

Getting Keyboard Input

Throughout these tutorials we've often attached event listeners to the `canvas`. While many events work, one that does not work by default is keyboard events.

To get keyboard events, set the `tabindex` of the canvas to 0 or more. Eg.

This ends up causing a new issue though. Anything that has a `tabindex` set will get highlighted when it has the focus. To fix that set its focus CSS outline to none

To demonstrate here are 3 canvases

and some css just for the last canvas

Let's attach the same event listeners to all of them

Notice you can't get the first canvas to accept keyboard input. The second canvas you can but it gets highlighted. The 3rd canvas has both solutions applied.

click here to open in a separate window

html

```
<canvas tabindex="0"></canvas>
```

css

```
canvas:focus {
  outline:none;
}
```

html

```
<canvas id="c1"></canvas>
<canvas id="c2" tabindex="0"></canvas>
<canvas id="c3" tabindex="1"></canvas>
```

css

```
#c3:focus {
    outline: none;
}
```

javascript

```
document.querySelectorAll('canvas').forEach((canvas) => {
  const ctx = canvas.getContext('2d');

  function draw(str) {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.textAlign = 'center';
    ctx.textBaseline = 'middle';
    ctx.fillText(str, canvas.width / 2, canvas.height / 2);
  }
  draw(canvas.id);

  canvas.addEventListener('focus', () => {
    draw('has focus press a key');
  });

  canvas.addEventListener('blur', () => {
    draw('lost focus');
  });

  canvas.addEventListener('keydown', (e) => {
    draw(`keyCode: ${e.keyCode}`);
  });
});
```

Making the Canvas Transparent

By default THREE.js makes the canvas opaque. If you want the canvas to be transparent pass in `alpha:true` when you create the `WebGLRenderer`

You probably also want to tell it that your results are **not** using premultiplied alpha

Three.js defaults to the canvas using `premultipliedAlpha: true` but defaults to materials outputting `premultipliedAlpha: false`.

If you'd like a better understanding of when and when not to use premultiplied alpha here's a good article on it.

In any case let's setup a simple example with a transparent canvas.

We applied the settings above to the example from the article on responsiveness. Let's also make the materials more transparent.

And let's add some HTML content

as well as some CSS to put the canvas in front

note that `pointer-events: none` makes the canvas invisible to the mouse and touch events so you can select the text beneath.

click here to open in a separate window

javascript

```
const canvas = document.querySelector('#c');
-const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
+const renderer = new THREE.WebGLRenderer({
+  canvas,
+  alpha: true,
+});
```

javascript

```
const canvas = document.querySelector('#c');
const renderer = new THREE.WebGLRenderer({
  canvas,
  alpha: true,
+  premultipliedAlpha: false,
});
```

javascript

```
function makeInstance(geometry, color, x) {
-  const material = new THREE.MeshPhongMaterial({color});
+  const material = new THREE.MeshPhongMaterial({
+    color,
+    opacity: 0.5,
+  });

...
```

html

```
<body>
  <canvas id="c"></canvas>
+  <div id="content">
+    <div>
+      <h1>Cubes-R-Us!</h1>
+      <p>We make the best cubes!</p>
+    </div>
+  </div>
</body>
```

css

```
body {
    margin: 0;
}
#c {
    width: 100%;
    height: 100%;
    display: block;
+    position: fixed;
+    left: 0;
+    top: 0;
+    z-index: 2;
+    pointer-events: none;
}
+#content {
+  font-size: 7vw;
+  font-family: sans-serif;
+  text-align: center;
+  width: 100%;
+  height: 100%;
+  display: flex;
+  justify-content: center;
+  align-items: center;
+}
```

Making your background a three.js animation

A common question is how to make a three.js animation be the background of a webpage.

There are 2 obvious ways.

- Set the canvas CSS `position` to `fixed` as in

You can basically see this exact solution on the previous example. Just set `z-index` to -1 and the cubes will appear behind the text.

A small disadvantage to this solution is your JavaScript must integrate with the page and if you have a complex page then you need to make sure none of the JavaScript in your three.js visualization conflict with the JavaScript doing other things in the page.

- Use an `iframe`

This is the solution used on [the front page of this site](#).

In your webpage just insert an iframe, for example

Then style the iframe to fill the window and be in the background which is basically the same code as we used above for the canvas except we also need to set `border` to `none` since iframes have a border by default.

[click here to open in a separate window](#)

```css
#c {
 position: fixed;
 left: 0;
 top: 0;
 ...
}
```

```html
<iframe id="background" src="responsive.html">
<div>
  Your content goes here.
</div>
```

```css
```

```
#background {
    position: fixed;
    width: 100%;
    height: 100%;
    left: 0;
    top: 0;
    z-index: -1;
    border: none;
    pointer-events: none;
}
```

Optimize Lots of Objects

GUIDE

Source: https://threejs.org/manual/en/optimize-lots-of-objects.html

A guide on optimizing three.js performance by merging many separate geometries into a single mesh, using the recreation of a WebGL Globe with demographic data as an example. Covers data loading, initial slow implementation, geometry merging with BufferGeometryUtils, and restoring per-object colors using vertex colors.

Optimize Lots of Objects

There are many ways to optimize things for three.js. One way is often referred to as *merging geometry*. Every `Mesh` you create and three.js represents 1 or more requests by the system to draw something. Drawing 2 things has more overhead than drawing 1 even if the results are the same so one way to optimize is to merge meshes.

Let's show an example of when this is a good solution for an issue. Let's re-create the WebGL Globe.

The first thing we need to do is get some data. The WebGL Globe said the data they use comes from SEDAC. Checking out the site I saw there was demographic data in a grid format. I downloaded the data at 60 minute resolution. Then I took a look at the data

It looks like this

```
ncols         360
nrows         145
xllcorner     -180
yllcorner     -60
cellsize      0.99999999999994
NODATA_value  -9999
-9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
...
```

There's a few lines that are like key/value pairs followed by lines with a value per grid point, one line for each row of data points.

To make sure we understand the data let's try to plot it in 2D.

First some code to load the text file

```
async function loadFile(url) {
  const res = await fetch(url);
  return res.text();
}
```

The code above returns a `Promise` with the contents of the file at `url`;

Then we need some code to parse the file

```
function parseData(text) {
  const data = [];
  const settings = {data};
  let max;
  let min;
  // split into lines
  text.split('\n').forEach((line) => {
    // split the line by whitespace
    const parts = line.trim().split(/\s+/);
    if (parts.length === 2) {
      // only 2 parts, must be a key/value pair
      settings[parts[0]] = parseFloat(parts[1]);
    } else if (parts.length > 2) {
      // more than 2 parts, must be data
      const values = parts.map((v) => {
        const value = parseFloat(v);
        if (value === settings.NODATA_value) {
          return undefined;
        }
        max = Math.max(max === undefined ? value : max, value);
        min = Math.min(min === undefined ? value : min, value);
        return value;
      });
      data.push(values);
    }
  });
  return Object.assign(settings, {min, max});
}
```

The code above returns an object with all the key/value pairs from the file as well as a `data` property with all the data in one large array and the `min` and `max` values found in the data.

Then we need some code to draw that data

```
function drawData(file) {
  const {min, max, data} = file;
  const range = max - min;
  const ctx = document.querySelector('canvas').getContext('2d');
  // make the canvas the same size as the data
  ctx.canvas.width = ncols;
  ctx.canvas.height = nrows;
  // but display it double size so it's not too small
  ctx.canvas.style.width = px(ncols * 2);
  ctx.canvas.style.height = px(nrows * 2);
  // fill the canvas to dark gray
  ctx.fillStyle = '#444';
  ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
  // draw each data point
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
      if (value === undefined) {
        return;
      }
      const amount = (value - min) / range;
      const hue = 1;
      const saturation = 1;
      const lightness = amount;
      ctx.fillStyle = hsl(hue, saturation, lightness);
      ctx.fillRect(lonNdx, latNdx, 1, 1);
    });
  });
}

function px(v) {
  return ${v | 0}px;
}

function hsl(h, s, l) {
  return hsl(${h * 360 | 0},${s * 100 | 0}%,${l * 100 | 0}%);
}
```

And finally gluing it all together

```
loadFile('resources/data/gpw/gpw_v4_basic_demographic_characteristics_rev10_a000_01
  .then(parseData)
  .then(drawData);
```

Gives us this result

click here to open in a separate window

So that seems to work.

Let's try it in 3D. Starting with the code from rendering on demand We'll make one box per data in the file.

First let's make a simple sphere with a texture of the world. Here's the texture

And the code to set it up.

```
{
  const loader = new THREE.TextureLoader();
  const texture = loader.load('resources/images/world.jpg', render);
  const geometry = new THREE.SphereGeometry(1, 64, 32);
  const material = new THREE.MeshBasicMaterial({map: texture});
  scene.add(new THREE.Mesh(geometry, material));
}
```

Notice the call to `render` when the texture has finished loading. We need this because we're rendering on demand instead of continuously so we need to render once when the texture is loaded.

Then we need to change the code that drew a dot per data point above to instead make a box per data point.

```
function addBoxes(file) {
  const {min, max, data} = file;
  const range = max - min;

  // make one box geometry
  const boxWidth = 1;
  const boxHeight = 1;
  const boxDepth = 1;
  const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
  // make it so it scales away from the positive Z axis
  geometry.applyMatrix4(new THREE.Matrix4().makeTranslation(0, 0, 0.5));

  // these helpers will make it easy to position the boxes
  // We can rotate the lon helper on its Y axis to the longitude
  const lonHelper = new THREE.Object3D();
  scene.add(lonHelper);
  // We rotate the latHelper on its X axis to the latitude
  const latHelper = new THREE.Object3D();
  lonHelper.add(latHelper);
  // The position helper moves the object to the edge of the sphere
  const positionHelper = new THREE.Object3D();
  positionHelper.position.z = 1;
  latHelper.add(positionHelper);

  const lonFudge = Math.PI * .5;
  const latFudge = Math.PI * -0.135;
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
      if (value === undefined) {
        return;
      }
      const amount = (value - min) / range;
      const material = new THREE.MeshBasicMaterial();
      const hue = THREE.MathUtils.lerp(0.7, 0.3, amount);
      const saturation = 1;
      const lightness = THREE.MathUtils.lerp(0.1, 1.0, amount);
      material.color.setHSL(hue, saturation, lightness);
      const mesh = new THREE.Mesh(geometry, material);
      scene.add(mesh);

      // adjust the helpers to point to the latitude and longitude
      lonHelper.rotation.y = THREE.MathUtils.degToRad(lonNdx + file.xllcorner) + lo
      latHelper.rotation.x = THREE.MathUtils.degToRad(latNdx + file.yllcorner) + la

      // use the world matrix of the position helper to
      // position this mesh.
      positionHelper.updateWorldMatrix(true, false);
      mesh.applyMatrix4(positionHelper.matrixWorld);

      mesh.scale.set(0.005, 0.005, THREE.MathUtils.lerp(0.01, 0.5, amount));
    });
  });
}
```

The code is mostly straight forward from our test drawing code.

We make one box and adjust its center so it scales away from positive Z. If we didn't do this it would scale from the center but we want them to grow away from the origin.

Of course we could also solve that by parenting the box to more `THREE.Object3D` objects like we covered in scene graphs but the more nodes we add to a scene graph the slower it gets.

We also setup this small hierarchy of nodes of `lonHelper`, `latHelper`, and `positionHelper`. We use these objects to compute a position around the sphere were to place the box.

Above the green bar represents `lonHelper` and is used to rotate toward longitude on the equator. The blue bar represents `latHelper` which is used to rotate to a latitude above or below the equator. The red sphere represents the offset that that `positionHelper` provides.

We could do all of the math manually to figure out positions on the globe but doing it this way leaves most of the math to the library itself so we don't need to deal with.

For each data point we create a `MeshBasicMaterial` and a `Mesh` and then we ask for the world matrix of the `positionHelper` and apply that to the new `Mesh`. Finally we scale the mesh at its new position.

Like above, we could also have created a `latHelper`, `lonHelper`, and `positionHelper` for every new box but that would be even slower.

There are up to 360x145 boxes we're going to create. That's up to 52000 boxes. Because some data points are marked as "NO_DATA" the actual number of boxes we're going to create is around 19000. If we added 3 extra helper objects per box that would be nearly 80000 scene graph nodes that THREE.js would have to compute positions for. By instead using one set of helpers to just position the meshes we save around 60000 operations.

A note about `lonFudge` and `latFudge`. `lonFudge` is π/2 which is a quarter of a turn. That makes sense. It just means the texture or texture coordinates start at a different offset around the globe. `latFudge` on the other hand I have no idea why it needs to be π * -0.135, that's just an amount that made the boxes line up with the texture.

The last thing we need to do is call our loader

```
loadFile('resources/data/gpw/gpw_v4_basic_demographic_characteristics_rev10_a000_01
  .then(parseData)
-  .then(drawData)
+  .then(addBoxes)
+  .then(render);
```

Once the data has finished loading and parsing then we need to render at least once since we're rendering on demand.

click here to open in a separate window

If you try to rotate the example above by dragging on the sample you'll likely notice it's slow.

We can check the framerate by opening the devtools and turning on the browser's frame rate meter.

On my machine I see a framerate under 20fps.

That doesn't feel very good to me and I suspect many people have slower machines which would make it even worse. We'd better look into optimizing.

For this particular problem we can merge all the boxes into a single geometry. We're currently drawing around 19000 boxes. By merging them into a single geometry we'd remove 18999 operations.

Here's the new code to merge the boxes into a single geometry.

```
function addBoxes(file) {
  const {min, max, data} = file;
  const range = max - min;

  // these helpers will make it easy to position the boxes
  // We can rotate the lon helper on its Y axis to the longitude
  const lonHelper = new THREE.Object3D();
  scene.add(lonHelper);
  // We rotate the latHelper on its X axis to the latitude
  const latHelper = new THREE.Object3D();
  lonHelper.add(latHelper);
  // The position helper moves the object to the edge of the sphere
  const positionHelper = new THREE.Object3D();
  positionHelper.position.z = 1;
  latHelper.add(positionHelper);
+  // Used to move the center of the box so it scales from the position Z axis
+  const originHelper = new THREE.Object3D();
+  originHelper.position.z = 0.5;
+  positionHelper.add(originHelper);

  const lonFudge = Math.PI * .5;
  const latFudge = Math.PI * -0.135;
+  const geometries = [];
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
      if (value === undefined) {
        return;
      }
      const amount = (value - min) / range;

+      const boxWidth = 1;
+      const boxHeight = 1;
+      const boxDepth = 1;
+      const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);

      // adjust the helpers to point to the latitude and longitude
      lonHelper.rotation.y = THREE.MathUtils.degToRad(lonNdx + file.xllcorner) + lo
      latHelper.rotation.x = THREE.MathUtils.degToRad(latNdx + file.yllcorner) + la

+      // use the world matrix of the origin helper to
+      // position this geometry
+      positionHelper.scale.set(0.005, 0.005, THREE.MathUtils.lerp(0.01, 0.5, amoun
+      originHelper.updateWorldMatrix(true, false);
+      geometry.applyMatrix4(originHelper.matrixWorld);
+
+      geometries.push(geometry);
    });
  });

+  const mergedGeometry = BufferGeometryUtils.mergeGeometries(
+      geometries, false);
+  const material = new THREE.MeshBasicMaterial({color:'red'});
+  const mesh = new THREE.Mesh(mergedGeometry, material);
+  scene.add(mesh);

}
```

Above we removed the code that was changing the box geometry's center point and are instead doing it by adding an `originHelper`. Before we were using the same geometry 19000 times. This time we are creating new geometry for every single box and since we are going to use `applyMatrix` to move the vertices of each box geometry we might as well do it once instead of twice.

At the end we pass an array of all the geometries to `BufferGeometryUtils.mergeGeometries` which will combined all of them into a single mesh.

We also need to include the `BufferGeometryUtils`

```
import * as BufferGeometryUtils from 'three/addons/utils/BufferGeometryUtils.js';
```

And now, at least on my machine, I get 60 frames per second

click here to open in a separate window

So that worked but because it's one mesh we only get one material which means we only get one color where as before we had a different color on each box. We can fix that by using vertex colors.

Vertex colors add a color per vertex. By setting all the colors of each vertex of each box to specific colors every box will have a different color.

```
+const color = new THREE.Color();

const lonFudge = Math.PI * .5;
const latFudge = Math.PI * -0.135;
const geometries = [];
data.forEach((row, latNdx) => {
  row.forEach((value, lonNdx) => {
    if (value === undefined) {
      return;
    }
    const amount = (value - min) / range;

    const boxWidth = 1;
    const boxHeight = 1;
    const boxDepth = 1;
    const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);

    // adjust the helpers to point to the latitude and longitude
    lonHelper.rotation.y = THREE.MathUtils.degToRad(lonNdx + file.xllcorner) + lonF
    latHelper.rotation.x = THREE.MathUtils.degToRad(latNdx + file.yllcorner) + latF

    // use the world matrix of the origin helper to
    // position this geometry
    positionHelper.scale.set(0.005, 0.005, THREE.MathUtils.lerp(0.01, 0.5, amount))
    originHelper.updateWorldMatrix(true, false);
    geometry.applyMatrix4(originHelper.matrixWorld);

+    // compute a color
+    const hue = THREE.MathUtils.lerp(0.7, 0.3, amount);
+    const saturation = 1;
+    const lightness = THREE.MathUtils.lerp(0.4, 1.0, amount);
+    color.setHSL(hue, saturation, lightness);
+    // get the colors as an array of values from 0 to 255
+    const rgb = color.toArray().map(v => v * 255);
+
+    // make an array to store colors for each vertex
+    const numVerts = geometry.getAttribute('position').count;
+    const itemSize = 3;  // r, g, b
+    const colors = new Uint8Array(itemSize * numVerts);
+
+    // copy the color into the colors array for each vertex
+    colors.forEach((v, ndx) => {
+      colors[ndx] = rgb[ndx % 3];
+    });
+
+    const normalized = true;
+    const colorAttrib = new THREE.BufferAttribute(colors, itemSize, normalized);
+    geometry.setAttribute('color', colorAttrib);

    geometries.push(geometry);
  });
});
```

The code above looks up the number or vertices needed by getting the `position` attribute from the geometry. We then create a `Uint8Array` to put the colors in. It then adds that as an attribute by calling `geometry.setAttribute`.

Lastly we need to tell three.js to use the vertex colors.

```
const mergedGeometry = BufferGeometryUtils.mergeGeometries(
    geometries, false);
-const material = new THREE.MeshBasicMaterial({color:'red'});
+const material = new THREE.MeshBasicMaterial({
+  vertexColors: true,
+});
const mesh = new THREE.Mesh(mergedGeometry, material);
scene.add(mesh);
```

And with that we get our colors back

click here to open in a separate window

Merging geometry is a common optimization technique. For example rather than 100 trees you might merge the trees into 1 geometry, a pile of individual rocks into a single geometry of rocks, a picket fence from individual pickets into one fence mesh. Another example in Minecraft it doesn't likely draw each cube individually but rather creates groups of merged cubes and also selectively removing faces that are never visible.

The problem with making everything one mesh though is it's no longer easy to move any part that was previously separate. Depending on our use case though there are creative solutions. We'll explore one in another article.

```text
ncols         360
 nrows         145
 xllcorner     -180
 yllcorner     -60
 cellsize      0.99999999999994
 NODATA_value  -9999
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
 9.241768 8.790958 2.095345 -9999 0.05114867 -9999 -9999 -9999 -9999 -999...
 1.287993 0.4395509 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999...
 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 -9999 ...
```

javascript

```
async function loadFile(url) {
  const res = await fetch(url);
  return res.text();
}
```

javascript

```
function parseData(text) {
  const data = [];
  const settings = {data};
  let max;
  let min;
  // split into lines
  text.split('\n').forEach((line) => {
    // split the line by whitespace
    const parts = line.trim().split(/\s+/);
    if (parts.length === 2) {
      // only 2 parts, must be a key/value pair
      settings[parts[0]] = parseFloat(parts[1]);
    } else if (parts.length > 2) {
      // more than 2 parts, must be data
      const values = parts.map((v) => {
        const value = parseFloat(v);
        if (value === settings.NODATA_value) {
          return undefined;
        }
        max = Math.max(max === undefined ? value : max, value);
        min = Math.min(min === undefined ? value : min, value);
        return value;
      });
      data.push(values);
    }
  });
  return Object.assign(settings, {min, max});
}
```

javascript

```
function drawData(file) {
  const {min, max, data} = file;
  const range = max - min;
  const ctx = document.querySelector('canvas').getContext('2d');
  // make the canvas the same size as the data
  ctx.canvas.width = ncols;
  ctx.canvas.height = nrows;
  // but display it double size so it's not too small
  ctx.canvas.style.width = px(ncols * 2);
  ctx.canvas.style.height = px(nrows * 2);
  // fill the canvas to dark gray
  ctx.fillStyle = '#444';
  ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
  // draw each data point
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
      if (value === undefined) {
        return;
      }
      const amount = (value - min) / range;
      const hue = 1;
      const saturation = 1;
      const lightness = amount;
      ctx.fillStyle = hsl(hue, saturation, lightness);
      ctx.fillRect(lonNdx, latNdx, 1, 1);
    });
  });
}

function px(v) {
  return ${v | 0}px;
}

function hsl(h, s, l) {
  return hsl(${h * 360 | 0},${s * 100 | 0}%,${l * 100 | 0}%);
}
```

javascript

```
loadFile('resources/data/gpw/gpw_v4_basic_demographic_characteristics_rev10_a000_01
  .then(parseData)
  .then(drawData);
```

javascript

```
{
  const loader = new THREE.TextureLoader();
  const texture = loader.load('resources/images/world.jpg', render);
  const geometry = new THREE.SphereGeometry(1, 64, 32);
  const material = new THREE.MeshBasicMaterial({map: texture});
  scene.add(new THREE.Mesh(geometry, material));
}
```

javascript

```
function addBoxes(file) {
  const {min, max, data} = file;
  const range = max - min;

  // make one box geometry
  const boxWidth = 1;
  const boxHeight = 1;
  const boxDepth = 1;
  const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
  // make it so it scales away from the positive Z axis
  geometry.applyMatrix4(new THREE.Matrix4().makeTranslation(0, 0, 0.5));

  // these helpers will make it easy to position the boxes
  // We can rotate the lon helper on its Y axis to the longitude
  const lonHelper = new THREE.Object3D();
  scene.add(lonHelper);
  // We rotate the latHelper on its X axis to the latitude
  const latHelper = new THREE.Object3D();
  lonHelper.add(latHelper);
  // The position helper moves the object to the edge of the sphere
  const positionHelper = new THREE.Object3D();
  positionHelper.position.z = 1;
  latHelper.add(positionHelper);

  const lonFudge = Math.PI * .5;
  const latFudge = Math.PI * -0.135;
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
      if (value === undefined) {
        return;
      }
      const amount = (value - min) / range;
      const material = new THREE.MeshBasicMaterial();
      const hue = THREE.MathUtils.lerp(0.7, 0.3, amount);
      const saturation = 1;
      const lightness = THREE.MathUtils.lerp(0.1, 1.0, amount);
      material.color.setHSL(hue, saturation, lightness);
      const mesh = new THREE.Mesh(geometry, material);
      scene.add(mesh);

      // adjust the helpers to point to the latitude and longitude
      lonHelper.rotation.y = THREE.MathUtils.degToRad(lonNdx + file.xllcorner) + lo
      latHelper.rotation.x = THREE.MathUtils.degToRad(latNdx + file.yllcorner) + la

      // use the world matrix of the position helper to
      // position this mesh.
      positionHelper.updateWorldMatrix(true, false);
      mesh.applyMatrix4(positionHelper.matrixWorld);

      mesh.scale.set(0.005, 0.005, THREE.MathUtils.lerp(0.01, 0.5, amount));
    });
  });
}
```

javascript

```
loadFile('resources/data/gpw/gpw_v4_basic_demographic_characteristics_rev10_a000_01
  .then(parseData)
-  .then(drawData)
+  .then(addBoxes)
+  .then(render);
```

javascript

```
function addBoxes(file) {
  const {min, max, data} = file;
  const range = max - min;

  // these helpers will make it easy to position the boxes
  // We can rotate the lon helper on its Y axis to the longitude
  const lonHelper = new THREE.Object3D();
  scene.add(lonHelper);
  // We rotate the latHelper on its X axis to the latitude
  const latHelper = new THREE.Object3D();
  lonHelper.add(latHelper);
  // The position helper moves the object to the edge of the sphere
  const positionHelper = new THREE.Object3D();
  positionHelper.position.z = 1;
  latHelper.add(positionHelper);
+  // Used to move the center of the box so it scales from the position Z axis
+  const originHelper = new THREE.Object3D();
+  originHelper.position.z = 0.5;
+  positionHelper.add(originHelper);

  const lonFudge = Math.PI * .5;
  const latFudge = Math.PI * -0.135;
+  const geometries = [];
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
      if (value === undefined) {
        return;
      }
      const amount = (value - min) / range;

+      const boxWidth = 1;
+      const boxHeight = 1;
+      const boxDepth = 1;
+      const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);

      // adjust the helpers to point to the latitude and longitude
      lonHelper.rotation.y = THREE.MathUtils.degToRad(lonNdx + file.xllcorner) + lo
      latHelper.rotation.x = THREE.MathUtils.degToRad(latNdx + file.yllcorner) + la

+      // use the world matrix of the origin helper to
+      // position this geometry
+      positionHelper.scale.set(0.005, 0.005, THREE.MathUtils.lerp(0.01, 0.5, amour
+      originHelper.updateWorldMatrix(true, false);
+      geometry.applyMatrix4(originHelper.matrixWorld);
+
+      geometries.push(geometry);
    });
  });

+  const mergedGeometry = BufferGeometryUtils.mergeGeometries(
+      geometries, false);
+  const material = new THREE.MeshBasicMaterial({color:'red'});
+  const mesh = new THREE.Mesh(mergedGeometry, material);
+  scene.add(mesh);

}
```

javascript

```
import * as BufferGeometryUtils from 'three/addons/utils/BufferGeometryUtils.js';
```

javascript

```
+const color = new THREE.Color();

const lonFudge = Math.PI * .5;
const latFudge = Math.PI * -0.135;
const geometries = [];
data.forEach((row, latNdx) => {
  row.forEach((value, lonNdx) => {
    if (value === undefined) {
      return;
    }
    const amount = (value - min) / range;

    const boxWidth = 1;
    const boxHeight = 1;
    const boxDepth = 1;
    const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);

    // adjust the helpers to point to the latitude and longitude
    lonHelper.rotation.y = THREE.MathUtils.degToRad(lonNdx + file.xllcorner) + lonF
    latHelper.rotation.x = THREE.MathUtils.degToRad(latNdx + file.yllcorner) + latF

    // use the world matrix of the origin helper to
    // position this geometry
    positionHelper.scale.set(0.005, 0.005, THREE.MathUtils.lerp(0.01, 0.5, amount))
    originHelper.updateWorldMatrix(true, false);
    geometry.applyMatrix4(originHelper.matrixWorld);

+    // compute a color
+    const hue = THREE.MathUtils.lerp(0.7, 0.3, amount);
+    const saturation = 1;
+    const lightness = THREE.MathUtils.lerp(0.4, 1.0, amount);
+    color.setHSL(hue, saturation, lightness);
+    // get the colors as an array of values from 0 to 255
+    const rgb = color.toArray().map(v => v * 255);
+
+    // make an array to store colors for each vertex
+    const numVerts = geometry.getAttribute('position').count;
+    const itemSize = 3;  // r, g, b
+    const colors = new Uint8Array(itemSize * numVerts);
+
+    // copy the color into the colors array for each vertex
+    colors.forEach((v, ndx) => {
+      colors[ndx] = rgb[ndx % 3];
+    });
+
+    const normalized = true;
+    const colorAttrib = new THREE.BufferAttribute(colors, itemSize, normalized);
+    geometry.setAttribute('color', colorAttrib);

    geometries.push(geometry);
  });
});
```

```
javascript
```

```
const mergedGeometry = BufferGeometryUtils.mergeGeometries(
    geometries, false);
-const material = new THREE.MeshBasicMaterial({color:'red'});
+const material = new THREE.MeshBasicMaterial({
+  vertexColors: true,
+});
const mesh = new THREE.Mesh(mergedGeometry, material);
scene.add(mesh);
```

Optimize Lots of Objects Animated

GUIDE

Source: https://threejs.org/manual/en/optimize-lots-of-objects-animated.html

A continuation of the article about optimizing lots of objects, this tutorial covers how to graph and animate between multiple sets of data using morphtargets in Three.js, including loading multiple data files, generating comparison datasets, building a UI to switch between them, and creating smooth transitions with a TweenManager.

Optimize Lots of Objects Animated

This article is a continuation of an article about optimizing lots of objects. If you haven't read that yet please read it before proceeding.

In the previous article we merged around 19000 cubes into a single geometry. This had the advantage that it optimized our drawing of 19000 cubes but it had the disadvantage of make it harder to move any individual cube.

Depending on what we are trying to accomplish there are different solutions. In this case let's graph multiple sets of data and animate between the sets.

The first thing we need to do is get multiple sets of data. Ideally we'd probably pre-process data offline but in this case let's load 2 sets of data and generate 2 more

Here's our old loading code

The code above will load all the files in `fileInfos` and when done each object in `fileInfos` will have a `file` property with the loaded file. `name` and `hueRange` we'll use later. `name` will be for a UI field. `hueRange` will be used to choose a range of hues to map over.

The two files above are apparently the number of men per area and the number of women per area as of 2010. Note, I have no idea if this data is correct but it's not important really. The important part is showing different sets of data.

Let's generate 2 more sets of data. One being the places where the number men are greater than the number of women and vice versa, the places where the number of women are greater than the number of men.

The first thing let's write a function that given a 2 dimensional array of arrays like we had before will map over it to generate a new 2 dimensional array of arrays

Like the normal `Array.map` function the `mapValues` function calls a function `fn` for each value in the array of arrays. It passes it the value and both the row and column indices.

Now let's make some code to generate a new file that is a comparison between 2 files

The code above uses `mapValues` to generate a new set of data that is a comparison based on the `compareFn` function passed in. It also tracks the `min` and `max` comparison results. Finally it makes a new file with all the same properties as `baseFile` except with a new `min`, `max` and `data`.

Then let's use that to make 2 new sets of data

Now let's generate a UI to select between these sets of data. First we need some UI html

and some CSS to make it appear in the top left area

Then we can go over each file and generate a set of merged boxes per set of data and an element which when hovered over will show that set and hide all others.

The one more change we need from the previous example is we need to make `addBoxes` take a `hueRange`

and with that we should be able to show 4 sets of data. Hover the mouse over the labels or touch them to switch sets

Note, there are a few strange data points that really stick out. I wonder what's up with those!??! In any case how do we animate between these 4 sets of data.

Lots of ideas.

- Just fade between them using `Material.opacity`

The problem with this solution is the cubes perfectly overlap which means there will be z-fighting issues. It's possible we could fix that by changing the depth function and using blending. We should probably look into it.

- Scale up the set we want to see and scale down the other sets

Because all the boxes have their origin at the center of the planet if we scale them below 1.0 they will sink into the planet. At first that sounds like a good idea but the issue is all the low height boxes will disappear almost immediately and not be replaced until the new data set scales up to 1.0. This makes the transition not very pleasant. We could maybe fix that with a fancy custom shader.

- Use Morphtargets

Morphtargets are a way were we supply multiple values for each vertex in the geometry and morph or lerp (linear interpolate) between them. Morphtargets are most commonly used for facial animation of 3D characters but that's not their only use.

Let's try morphtargets.

We'll still make a geometry for each set of data but we'll then extract the `position` attribute from each one and use them as morphtargets.

First let's change `addBoxes` to just make and return the merged geometry.

There's one more thing we need to do here though. Morphtargets are required to all have exactly the same number of vertices. Vertex #123 in one target needs have a corresponding Vertex #123 in all other targets. But, as it is now different data sets might have some data points with no data so no box will be generated for that point which would mean no corresponding vertices for another set. So, we need to check across all data sets and either always generate something if there is data in any set or, generate nothing if there is data missing in any set. Let's do the latter.

Now we'll change the code that was calling `addBoxes` to use `makeBoxes` and setup morphtargets

Above we make geometry for each data set, use the first one as the base, then get a `position` attribute from each geometry and add it as a morphtarget to the base geometry for `position`.

Now we need to change how we're showing and hiding the various data sets. Instead of showing or hiding a mesh we need to change the influence of the morphtargets. For the data set we want to see we need to have an influence of 1 and for all the ones we don't want to see to we need to have an influence of 0.

We could just set them to 0 or 1 directly but if we did that we wouldn't see any animation, it would just snap which would be no different than what we already have. We could also write some custom animation code which would be easy but because the original webgl globe uses an animation library let's use the same one here.

We need to include the library

And then create a `Tween` to animate the influences.

We're also suppose to call `TWEEN.update` every frame inside our render loop but that points out a problem. "tween.js" is designed for continuous rendering but we are rendering on demand. We could switch to continuous rendering but it's sometimes nice to only render on demand as it well stop using the user's power when nothing is happening so let's see if we can make it animate on demand.

We'll make a `TweenManager` to help. We'll use it to create the `Tween`s and track them. It will have an `update` method that will return `true` if we need to call it again and `false` if all the animations are finished.

To use it we'll create one

We'll use it to create our `Tween`s.

Then we'll update our render loop to update the tweens and keep rendering if there are still animations running.

And with that we should be animating between data sets.

I hope going through this was helpful. Using morphtargets is a common technique to move lots of objects. As an example we could give every cube a random place in another target and morph from that to their first positions on the globe. That might be a cool way to introduce the globe.

Next you might be interested in adding labels to a globe which is covered in Aligning HTML Elements to 3D.

Note: We could try to just graph percent of men or percent of women or the raw difference but based on how we are displaying the info, cubes that grow from the surface of the earth, we'd prefer most cubes to be low. If we used one of these other comparisons most cubes would be about 1/2 their maximum height which would not make a good visualization. Feel free to change the `amountGreaterThan` from `Math.max(a - b, 0)` to something like `(a - b)` "raw difference" or `a / (a + b)` "percent" and you'll see what I mean.

javascript

```
loadFile('resources/data/gpw/gpw_v4_basic_demographic_characteristics_rev10_a000_01
  .then(parseData)
  .then(addBoxes)
  .then(render);
```

javascript

```
async function loadData(info) {
  const text = await loadFile(info.url);
  info.file = parseData(text);
}

async function loadAll() {
  const fileInfos = [
    {name: 'men',   hueRange: [0.7, 0.3], url: 'resources/data/gpw/gpw_v4_basic_dem
    {name: 'women', hueRange: [0.9, 1.1], url: 'resources/data/gpw/gpw_v4_basic_dem
  ];

  await Promise.all(fileInfos.map(loadData));

  ...
}
loadAll();
```

javascript

```
function mapValues(data, fn) {
  return data.map((row, rowNdx) => {
    return row.map((value, colNdx) => {
      return fn(value, rowNdx, colNdx);
    });
  });
}
```

javascript

```
function makeDiffFile(baseFile, otherFile, compareFn) {
  let min;
  let max;
  const baseData = baseFile.data;
  const otherData = otherFile.data;
  const data = mapValues(baseData, (base, rowNdx, colNdx) => {
    const other = otherData[rowNdx][colNdx];
      if (base === undefined || other === undefined) {
        return undefined;
      }
      const value = compareFn(base, other);
      min = Math.min(min === undefined ? value : min, value);
      max = Math.max(max === undefined ? value : max, value);
      return value;
  });
  // make a copy of baseFile and replace min, max, and data
  // with the new data
  return {...baseFile, min, max, data};
}
```

javascript

```
{
  const menInfo = fileInfos[0];
  const womenInfo = fileInfos[1];
  const menFile = menInfo.file;
  const womenFile = womenInfo.file;

  function amountGreaterThan(a, b) {
    return Math.max(a - b, 0);
  }
  fileInfos.push({
    name: '>50%men',
    hueRange: [0.6, 1.1],
    file: makeDiffFile(menFile, womenFile, (men, women) => {
      return amountGreaterThan(men, women);
    }),
  });
  fileInfos.push({
    name: '>50% women',
    hueRange: [0.0, 0.4],
    file: makeDiffFile(womenFile, menFile, (women, men) => {
      return amountGreaterThan(women, men);
    }),
  });
}
```

html

```
<body>
  <canvas id="c"></canvas>
+  <div id="ui"></div>
</body>
```

css

```
#ui {
  position: absolute;
  left: 1em;
  top: 1em;
}
#ui>div {
  font-size: 20pt;
  padding: 1em;
  display: inline-block;
}
#ui>div.selected {
  color: red;
}
```

javascript

```
// show the selected data, hide the rest
function showFileInfo(fileInfos, fileInfo) {
  fileInfos.forEach((info) => {
    const visible = fileInfo === info;
    info.root.visible = visible;
    info.elem.className = visible ? 'selected' : '';
  });
  requestRenderIfNotRequested();
}

const uiElem = document.querySelector('#ui');
fileInfos.forEach((info) => {
  const boxes = addBoxes(info.file, info.hueRange);
  info.root = boxes;
  const div = document.createElement('div');
  info.elem = div;
  div.textContent = info.name;
  uiElem.appendChild(div);
  div.addEventListener('mouseover', () => {
    showFileInfo(fileInfos, info);
  });
});
// show the first set of data
showFileInfo(fileInfos, fileInfos[0]);
```

javascript

```
-function addBoxes(file) {
+function addBoxes(file, hueRange) {

  ...

    // compute a color
-    const hue = THREE.MathUtils.lerp(0.7, 0.3, amount);
+    const hue = THREE.MathUtils.lerp(...hueRange, amount);

  ...
```

javascript

```
-function addBoxes(file, hueRange) {
+function makeBoxes(file, hueRange) {
  const {min, max, data} = file;
  const range = max - min;

  ...

-  const mergedGeometry = BufferGeometryUtils.mergeGeometries(
-      geometries, false);
-  const material = new THREE.MeshBasicMaterial({
-    vertexColors: true,
-  });
-  const mesh = new THREE.Mesh(mergedGeometry, material);
-  scene.add(mesh);
-  return mesh;
+  return BufferGeometryUtils.mergeGeometries(
+     geometries, false);
}
```

javascript

```
+function dataMissingInAnySet(fileInfos, latNdx, lonNdx) {
+  for (const fileInfo of fileInfos) {
+    if (fileInfo.file.data[latNdx][lonNdx] === undefined) {
+      return true;
+    }
+  }
+  return false;
+}

-function makeBoxes(file, hueRange) {
+function makeBoxes(file, hueRange, fileInfos) {
  const {min, max, data} = file;
  const range = max - min;

  ...

  const geometries = [];
  data.forEach((row, latNdx) => {
    row.forEach((value, lonNdx) => {
+      if (dataMissingInAnySet(fileInfos, latNdx, lonNdx)) {
+        return;
+      }
      const amount = (value - min) / range;

  ...
```

javascript

```
+// make geometry for each data set
+const geometries = fileInfos.map((info) => {
+  return makeBoxes(info.file, info.hueRange, fileInfos);
+});
+
+// use the first geometry as the base
+// and add all the geometries as morphtargets
+const baseGeometry = geometries[0];
+baseGeometry.morphAttributes.position = geometries.map((geometry, ndx) => {
+  const attribute = geometry.getAttribute('position');
+  const name = target${ndx};
+  attribute.name = name;
+  return attribute;
+});
+baseGeometry.morphAttributes.color = geometries.map((geometry, ndx) => {
+  const attribute = geometry.getAttribute('color');
+  const name = target${ndx};
+  attribute.name = name;
+  return attribute;
+});
+const material = new THREE.MeshBasicMaterial({
+  vertexColors: true,
+});
+const mesh = new THREE.Mesh(baseGeometry, material);
+scene.add(mesh);

const uiElem = document.querySelector('#ui');
fileInfos.forEach((info) => {
-  const boxes = addBoxes(info.file, info.hueRange);
-  info.root = boxes;
  const div = document.createElement('div');
  info.elem = div;
  div.textContent = info.name;
  uiElem.appendChild(div);
  function show() {
    showFileInfo(fileInfos, info);
  }
  div.addEventListener('mouseover', show);
  div.addEventListener('touchstart', show);
});
// show the first set of data
showFileInfo(fileInfos, fileInfos[0]);
```

javascript

```
import * as THREE from 'three';
import * as BufferGeometryUtils from 'three/addons/utils/BufferGeometryUtils.js';
import {OrbitControls} from 'three/addons/controls/OrbitControls.js';
+import TWEEN from 'three/addons/libs/tween.module.js';
```

javascript

```
// show the selected data, hide the rest
function showFileInfo(fileInfos, fileInfo) {
+  const targets = {};
-  fileInfos.forEach((info) => {
+  fileInfos.forEach((info, i) => {
    const visible = fileInfo === info;
-    info.root.visible = visible;
    info.elem.className = visible ? 'selected' : '';
+    targets[i] = visible ? 1 : 0;
  });
+  const durationInMs = 1000;
+  new TWEEN.Tween(mesh.morphTargetInfluences)
+    .to(targets, durationInMs)
+    .start();
  requestRenderIfNotRequested();
}
```

javascript

```
class TweenManger {
  constructor() {
    this.numTweensRunning = 0;
  }
  _handleComplete() {
    --this.numTweensRunning;
    console.assert(this.numTweensRunning >= 0);
  }
  createTween(targetObject) {
    const self = this;
    ++this.numTweensRunning;
    let userCompleteFn = () => {};
    // create a new tween and install our own onComplete callback
    const tween = new TWEEN.Tween(targetObject).onComplete(function(...args) {
      self._handleComplete();
      userCompleteFn.call(this, ...args);
    });
    // replace the tween's onComplete function with our own
    // so we can call the user's callback if they supply one.
    tween.onComplete = (fn) => {
      userCompleteFn = fn;
      return tween;
    };
    return tween;
  }
  update() {
    TWEEN.update();
    return this.numTweensRunning > 0;
  }
}
```

javascript

```
function main() {
  const canvas = document.querySelector('#c');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
+  const tweenManager = new TweenManger();

  ...
```

javascript

```
// show the selected data, hide the rest
function showFileInfo(fileInfos, fileInfo) {
  const targets = {};
  fileInfos.forEach((info, i) => {
    const visible = fileInfo === info;
    info.elem.className = visible ? 'selected' : '';
    targets[i] = visible ? 1 : 0;
  });
  const durationInMs = 1000;
-  new TWEEN.Tween(mesh.morphTargetInfluences)
+  tweenManager.createTween(mesh.morphTargetInfluences)
    .to(targets, durationInMs)
    .start();
  requestRenderIfNotRequested();
}
```

javascript

```
function render() {
  renderRequested = false;

  if (resizeRendererToDisplaySize(renderer)) {
    const canvas = renderer.domElement;
    camera.aspect = canvas.clientWidth / canvas.clientHeight;
    camera.updateProjectionMatrix();
  }

+  if (tweenManager.update()) {
+    requestRenderIfNotRequested();
+  }

  controls.update();
  renderer.render(scene, camera);
}
render();
```

OffscreenCanvas

GUIDE

Source: https://threejs.org/manual/en/offscreencanvas.html

A tutorial on using OffscreenCanvas with Three.js to offload rendering work to web workers, including basic setup, picking, OrbitControls integration, and fallback for browsers without OffscreenCanvas support.

OffscreenCanvas

`OffscreenCanvas` is a relatively new browser feature currently only available in Chrome but apparently coming to other browsers. `OffscreenCanvas` allows a web worker to render to a canvas. This is a way to offload heavy work, like rendering a complex 3D scene, to a web worker so as not to slow down the responsiveness of the browser. It also means data is loaded and parsed in the worker so possibly less jank while the page loads.

Getting *started* using it is pretty straight forward. Let's port the 3 spinning cube example from the article on responsiveness.

Workers generally have their code separated into another script file whereas most of the examples on this site have had their scripts embedded into the HTML file of the page they are on.

In our case we'll make a file called `offscreencanvas-cubes.js` and copy all the JavaScript from the responsive example into it. We'll then make the changes needed for it to run in a worker.

We still need some JavaScript in our HTML file. The first thing we need to do there is look up the canvas and then transfer control of that canvas to be offscreen by calling `canvas.transferControlToOffscreen`.

```javascript
function main() {
  const canvas = document.querySelector('#c');
  const offscreen = canvas.transferControlToOffscreen();

  ...
```

Starting the Worker

We can then start our worker with `new Worker(pathToScript, {type: 'module'})`. and pass the `offscreen` object to it.

```javascript
```

```
function main() {
  const canvas = document.querySelector('#c');
  const offscreen = canvas.transferControlToOffscreen();
  const worker = new Worker('offscreencanvas-cubes.js', {type: 'module'});
  worker.postMessage({type: 'main', canvas: offscreen}, [offscreen]);
}
main();
```

Worker Message Communication

It's important to note that workers can't access the `DOM`. They can't look at HTML elements nor can they receive mouse events or keyboard events. The only thing they can generally do is respond to messages sent to them and send messages back to the page.

To send a message to a worker we call `worker.postMessage` and pass it 1 or 2 arguments. The first argument is a JavaScript object that will be cloned and sent to the worker. The second argument is an optional array of objects that are part of the first object that we want *transferred* to the worker. These objects will not be cloned. Instead they will be *transferred* and will cease to exist in the main page. Cease to exist is the probably the wrong description, rather they are neutered. Only certain types of objects can be transferred instead of cloned. They include `OffscreenCanvas` so once transferred the `offscreen` object back in the main page is useless.

Workers receive messages from their `onmessage` handler. The object we passed to `postMessage` arrives on `event.data` passed to the `onmessage` handler on the worker. The code above declares a `type: 'main'` in the object it passes to the worker. This object has no meaning to the browser. It's entirely for our own usage. We'll make a handler that based on `type` calls a different function in the worker. Then we can add functions as needed and easily call them from the main page.

javascript

```
const handlers = {
  main,
};

self.onmessage = function(e) {
  const fn = handlers[e.data.type];
  if (typeof fn !== 'function') {
    throw new Error('no handler for type: ' + e.data.type);
  }
  fn(e.data);
};
```

Adapting the Main Function for Worker

You can see above we just look up the handler based on the `type` pass it the `data` that was sent from the main page.

So now we just need to start changing the `main` we pasted into `offscreencanvas-cubes.js` from the responsive article.

Instead of looking up the canvas from the DOM we'll receive it from the event data.

javascript

```
-function main() {
-  const canvas = document.querySelector('#c');
+function main(data) {
+  const {canvas} = data;
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});

  ...
```

Handling Size Changes

Remembering that workers can't see the DOM at all the first problem we run into is `resizeRendererToDisplaySize` can't look at `canvas.clientWidth` and `canvas.clientHeight` as those are DOM values. Here's the original code

javascript

```
function resizeRendererToDisplaySize(renderer) {
  const canvas = renderer.domElement;
  const width = canvas.clientWidth;
  const height = canvas.clientHeight;
  const needResize = canvas.width !== width || canvas.height !== height;
  if (needResize) {
    renderer.setSize(width, height, false);
  }
  return needResize;
}
```

Instead we'll need to send sizes as they change to the worker. So, let's add some global state and keep the width and height there.

javascript

```
const state = {
  width: 300,  // canvas default
  height: 150,  // canvas default
};
```

Then let's add a `'size'` handler to update those values.

```javascript
+function size(data) {
+  state.width = data.width;
+  state.height = data.height;
+}

const handlers = {
  main,
+  size,
};
```

Now we can change `resizeRendererToDisplaySize` to use `state.width` and `state.height`

```javascript
function resizeRendererToDisplaySize(renderer) {
  const canvas = renderer.domElement;
-  const width = canvas.clientWidth;
-  const height = canvas.clientHeight;
+  const width = state.width;
+  const height = state.height;
  const needResize = canvas.width !== width || canvas.height !== height;
  if (needResize) {
    renderer.setSize(width, height, false);
  }
  return needResize;
}
```

and where we compute the aspect we need similar changes

```javascript
function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
-    camera.aspect = canvas.clientWidth / canvas.clientHeight;
+    camera.aspect = state.width / state.height;
    camera.updateProjectionMatrix();
  }

  ...
```

Back in the main page we'll send a `size` event anytime the page changes size.

```javascript
```

```
const worker = new Worker('offscreencanvas-picking.js', {type: 'module'});
worker.postMessage({type: 'main', canvas: offscreen}, [offscreen]);

+function sendSize() {
+  worker.postMessage({
+    type: 'size',
+    width: canvas.clientWidth,
+    height: canvas.clientHeight,
+  });
+}
+
+window.addEventListener('resize', sendSize);
+sendSize();
```

We also call it once to send the initial size.

And with just those few changes, assuming your browser fully supports `OffscreenCanvas` it should work. Before we run it though let's check if the browser actually supports `OffscreenCanvas` and if not display an error. First let's add some HTML to display the error.

html

```
<body>
  <canvas id="c"></canvas>
+  <div id="noOffscreenCanvas" style="display:none;">
+    <div>no OffscreenCanvas support</div>
+  </div>
</body>
```

and some CSS for that

css

```
#noOffscreenCanvas {
    display: flex;
    width: 100%;
    height: 100%;
    align-items: center;
    justify-content: center;
    background: red;
    color: white;
}
```

and then we can check for the existence of `transferControlToOffscreen` to see if the browser supports `OffscreenCanvas`

javascript

```
function main() {
  const canvas = document.querySelector('#c');
+  if (!canvas.transferControlToOffscreen) {
+    canvas.style.display = 'none';
+    document.querySelector('#noOffscreenCanvas').style.display = '';
+    return;
+  }
  const offscreen = canvas.transferControlToOffscreen();
  const worker = new Worker('offscreencanvas-picking.js', {type: 'module'});
  worker.postMessage({type: 'main', canvas: offscreen}, [offscreen]);

  ...
```

and with that, if your browser supports `OffscreenCanvas` this example should work

click here to open in a separate window

So that's great but since not every browser supports `OffscreenCanvas` at the moment let's change the code to work with both `OffscreenCanvas` and if not then fallback to using the canvas in the main page like normal.

> *As an aside, if you need OffscreenCanvas to make your page responsive then it's not clear what the point of having a fallback is. Maybe based on if you end up running on the main page or in a worker you might adjust the amount of work done so that when running in a worker you can do more than when running in the main page. What you do is really up to you.*

Adding Fallback Support

The first thing we should probably do is separate out the three.js code from the code that is specific to the worker. That way we can use the same code on both the main page and the worker. In other words we will now have 3 files

- our html file.

`threejs-offscreencanvas-w-fallback.html`

- a JavaScript that contains our three.js code.

`shared-cubes.js`

- our worker support code

`offscreencanvas-worker-cubes.js`

`shared-cubes.js` and `offscreencanvas-worker-cubes.js` are basically the split of our previous `offscreencanvas-cubes.js` file. First we copy all of `offscreencanvas-cubes.js` to `shared-cube.js`. Then we rename `main` to `init` since we already have a `main` in our HTML file and we need to export `init` and `state`

javascript

```
import * as THREE from 'three';

-const state = {
+export const state = {
  width: 300,   // canvas default
  height: 150,  // canvas default
};

-function main(data) {
+export function init(data) {
  const {canvas} = data;
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});
```

and cut out the just the non three.js relates parts

javascript

```
-function size(data) {
-  state.width = data.width;
-  state.height = data.height;
-}
-
-const handlers = {
-  main,
-  size,
-};
-
-self.onmessage = function(e) {
-  const fn = handlers[e.data.type];
-  if (typeof fn !== 'function') {
-    throw new Error('no handler for type: ' + e.data.type);
-  }
-  fn(e.data);
-};
```

Then we copy those parts we just deleted to `offscreencanvas-worker-cubes.js` and import `shared-cubes.js` as well as call `init` instead of `main`.

javascript

```
import {init, state} from './shared-cubes.js';

function size(data) {
  state.width = data.width;
  state.height = data.height;
}

const handlers = {
-  main,
+  init,
  size,
};

self.onmessage = function(e) {
  const fn = handlers[e.data.type];
  if (typeof fn !== 'function') {
    throw new Error('no handler for type: ' + e.data.type);
  }
  fn(e.data);
};
```

Similarly we need to include `shared-cubes.js` in the main page

```
html
```

```
<script type="module">
+import {init, state} from './shared-cubes.js';
```

We can remove the HTML and CSS we added previously

```
html
```

```
<body>
  <canvas id="c"></canvas>
-  <div id="noOffscreenCanvas" style="display:none;">
-    <div>no OffscreenCanvas support</div>
-  </div>
</body>
```

and some CSS for that

```
css
```

```
-#noOffscreenCanvas {
-    display: flex;
-    width: 100%;
-    height: 100%;
-    align-items: center;
-    justify-content: center;
-    background: red;
-    color: white;
-}
```

Then let's change the code in the main page to call one start function or another depending on if the browser supports `OffscreenCanvas`.

javascript

```
function main() {
  const canvas = document.querySelector('#c');
-  if (!canvas.transferControlToOffscreen) {
-    canvas.style.display = 'none';
-    document.querySelector('#noOffscreenCanvas').style.display = '';
-    return;
-  }
-  const offscreen = canvas.transferControlToOffscreen();
-  const worker = new Worker('offscreencanvas-picking.js', {type: 'module'});
-  worker.postMessage({type: 'main', canvas: offscreen}, [offscreen]);
+  if (canvas.transferControlToOffscreen) {
+    startWorker(canvas);
+  } else {
+    startMainPage(canvas);
+  }
  ...
```

We'll move all the code we had to setup the worker inside `startWorker`

javascript

```
function startWorker(canvas) {
  const offscreen = canvas.transferControlToOffscreen();
  const worker = new Worker('offscreencanvas-worker-cubes.js', {type: 'module'});
  worker.postMessage({type: 'main', canvas: offscreen}, [offscreen]);

  function sendSize() {
    worker.postMessage({
      type: 'size',
      width: canvas.clientWidth,
      height: canvas.clientHeight,
    });
  }

  window.addEventListener('resize', sendSize);
  sendSize();

  console.log('using OffscreenCanvas');
}
```

and send `init` instead of `main`

javascript

```
-  worker.postMessage({type: 'main', canvas: offscreen}, [offscreen]);
+  worker.postMessage({type: 'init', canvas: offscreen}, [offscreen]);
```

for starting in the main page we can do this

javascript

```
function startMainPage(canvas) {
  init({canvas});

  function sendSize() {
    state.width = canvas.clientWidth;
    state.height = canvas.clientHeight;
  }
  window.addEventListener('resize', sendSize);
  sendSize();

  console.log('using regular canvas');
}
```

and with that our example will run either in an OffscreenCanvas or fallback to running in the main page.

click here to open in a separate window

So that was relatively easy. Let's try picking. We'll take some code from the `RayCaster` example from the article on picking and make it work offscreen.

Let's copy the `shared-cube.js` to `shared-picking.js` and add the picking parts. We copy in the `PickHelper`

javascript

```
class PickHelper {
  constructor() {
    this.raycaster = new THREE.Raycaster();
    this.pickedObject = null;
    this.pickedObjectSavedColor = 0;
  }
  pick(normalizedPosition, scene, camera, time) {
    // restore the color if there is a picked object
    if (this.pickedObject) {
      this.pickedObject.material.emissive.setHex(this.pickedObjectSavedColor);
      this.pickedObject = undefined;
    }

    // cast a ray through the frustum
    this.raycaster.setFromCamera(normalizedPosition, camera);
    // get the list of objects the ray intersected
    const intersectedObjects = this.raycaster.intersectObjects(scene.children);
    if (intersectedObjects.length) {
      // pick the first object. It's the closest one
      this.pickedObject = intersectedObjects[0].object;
      // save its color
      this.pickedObjectSavedColor = this.pickedObject.material.emissive.getHex();
      // set its emissive color to flashing red/yellow
      this.pickedObject.material.emissive.setHex((time * 8) % 2 > 1 ? 0xFFFF00 : 0x
    }
  }
}

const pickPosition = {x: 0, y: 0};
const pickHelper = new PickHelper();
```

Picking with OffscreenCanvas

We updated `pickPosition` from the mouse like this

javascript

```
function getCanvasRelativePosition(event) {
  const rect = canvas.getBoundingClientRect();
  return {
    x: (event.clientX - rect.left) * canvas.width  / rect.width,
    y: (event.clientY - rect.top ) * canvas.height / rect.height,
  };
}

function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
  pickPosition.x = (pos.x / canvas.width ) *  2 - 1;
  pickPosition.y = (pos.y / canvas.height) * -2 + 1;  // note we flip Y
}
window.addEventListener('mousemove', setPickPosition);
```

A worker can't read the mouse position directly so just like the size code let's send a message with the mouse position. Like the size code we'll send the mouse position and update `pickPosition`

javascript

```
function size(data) {
  state.width = data.width;
  state.height = data.height;
}

+function mouse(data) {
+  pickPosition.x = data.x;
+  pickPosition.y = data.y;
+}

const handlers = {
  init,
+  mouse,
  size,
};

self.onmessage = function(e) {
  const fn = handlers[e.data.type];
  if (typeof fn !== 'function') {
    throw new Error('no handler for type: ' + e.data.type);
  }
  fn(e.data);
};
```

Back in our main page we need to add code to pass the mouse to the worker or the main page.

javascript

```
+let sendMouse;

function startWorker(canvas) {
  const offscreen = canvas.transferControlToOffscreen();
  const worker = new Worker('offscreencanvas-worker-picking.js', {type: 'module'});
  worker.postMessage({type: 'init', canvas: offscreen}, [offscreen]);

+  sendMouse = (x, y) => {
+    worker.postMessage({
+      type: 'mouse',
+      x,
+      y,
+    });
+  };

  function sendSize() {
    worker.postMessage({
      type: 'size',
      width: canvas.clientWidth,
      height: canvas.clientHeight,
    });
  }

  window.addEventListener('resize', sendSize);
  sendSize();

  console.log('using OffscreenCanvas');  /* eslint-disable-line no-console */
}

function startMainPage(canvas) {
  init({canvas});

+  sendMouse = (x, y) => {
+    pickPosition.x = x;
+    pickPosition.y = y;
+  };

  function sendSize() {
    state.width = canvas.clientWidth;
    state.height = canvas.clientHeight;
  }
  window.addEventListener('resize', sendSize);
  sendSize();

  console.log('using regular canvas');  /* eslint-disable-line no-console */
}
```

Then we can copy in all the mouse handling code to the main page and make just minor changes to use `sendMouse`

```
javascript
```

```
function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
-  pickPosition.x = (pos.x / canvas.clientWidth ) *  2 - 1;
-  pickPosition.y = (pos.y / canvas.clientHeight) * -2 + 1;  // note we flip Y
+  sendMouse(
+      (pos.x / canvas.clientWidth ) *  2 - 1,
+      (pos.y / canvas.clientHeight) * -2 + 1);  // note we flip Y
}

function clearPickPosition() {
  // unlike the mouse which always has a position
  // if the user stops touching the screen we want
  // to stop picking. For now we just pick a value
  // unlikely to pick something
-  pickPosition.x = -100000;
-  pickPosition.y = -100000;
+  sendMouse(-100000, -100000);
}
window.addEventListener('mousemove', setPickPosition);
window.addEventListener('mouseout', clearPickPosition);
window.addEventListener('mouseleave', clearPickPosition);

window.addEventListener('touchstart', (event) => {
  // prevent the window from scrolling
  event.preventDefault();
  setPickPosition(event.touches[0]);
}, {passive: false});

window.addEventListener('touchmove', (event) => {
  setPickPosition(event.touches[0]);
});

window.addEventListener('touchend', clearPickPosition);
```

and with that picking should be working with `OffscreenCanvas`.

click here to open in a separate window

Let's take it one more step and add in the `OrbitControls`. This will be little more involved. The `OrbitControls` use the DOM pretty extensively checking the mouse, touch events, and the keyboard.

Unlike our code so far we can't really use a global `state` object without re-writing all the OrbitControls code to work with it. The OrbitControls take an `HTMLElement` to which they attach most of the DOM events they use. Maybe we could pass in our own object that has the same API surface as a DOM element. We only need to support the features the OrbitControls need.

Digging through the OrbitControls source code it looks like we need to handle the following events.

- contextmenu
- pointerdown
- pointermove
- pointerup
- touchstart
- touchmove
- touchend
- wheel
- keydown

For the pointer events we need the `ctrlKey`, `metaKey`, `shiftKey`, `button`, `pointerType`, `clientX`, `clientY`, `pageX`, and `pageY`, properties.

For the keydown events we need the `ctrlKey`, `metaKey`, `shiftKey`, and `keyCode` properties.

For the wheel event we only need the `deltaY` property.

And for the touch events we only need `pageX` and `pageY` from the `touches` property.

So, let's make a proxy object pair. One part will run in the main page, get all those events, and pass on the relevant property values to the worker. The other part will run in the worker, receive those events and pass them on using events that have the same structure as the original DOM events so the OrbitControls won't be able to tell the difference.

Here's the code for the worker part.

OrbitControls with OffscreenCanvas

```
javascript
```

```
import {EventDispatcher} from 'three';

class ElementProxyReceiver extends EventDispatcher {
  constructor() {
    super();
  }
  handleEvent(data) {
    this.dispatchEvent(data);
  }
}
```

All it does is if it receives a message it dispatches it. It inherits from `EventDispatcher` which provides methods like `addEventListener` and `removeEventListener` just like a DOM element so if we pass it to the OrbitControls it should work.

`ElementProxyReceiver` handles 1 element. In our case we only need one but it's best to think head so lets make a manager to manage more than one of them.

javascript

```
class ProxyManager {
  constructor() {
    this.targets = {};
    this.handleEvent = this.handleEvent.bind(this);
  }
  makeProxy(data) {
    const {id} = data;
    const proxy = new ElementProxyReceiver();
    this.targets[id] = proxy;
  }
  getProxy(id) {
    return this.targets[id];
  }
  handleEvent(data) {
    this.targets[data.id].handleEvent(data.data);
  }
}
```

We can make a instance of `ProxyManager` and call its `makeProxy` method with an id which will make an `ElementProxyReceiver` that responds to messages with that id.

Let's hook it up to our worker's message handler.

javascript

```
const proxyManager = new ProxyManager();

function start(data) {
  const proxy = proxyManager.getProxy(data.canvasId);
  init({
    canvas: data.canvas,
    inputElement: proxy,
  });
}

function makeProxy(data) {
  proxyManager.makeProxy(data);
}

...

const handlers = {
-  init,
-  mouse,
+  start,
+  makeProxy,
+  event: proxyManager.handleEvent,
   size,
};

self.onmessage = function(e) {
  const fn = handlers[e.data.type];
  if (typeof fn !== 'function') {
    throw new Error('no handler for type: ' + e.data.type);
  }
  fn(e.data);
};
```

In our shared three.js code we need to import the `OrbitControls` and set them up.

javascript

```
import * as THREE from 'three';
+import {OrbitControls} from 'three/addons/controls/OrbitControls.js';

export function init(data) {
-  const {canvas} = data;
+  const {canvas, inputElement} = data;
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas});

+  const controls = new OrbitControls(camera, inputElement);
+  controls.target.set(0, 0, 0);
+  controls.update();
```

Notice we're passing the OrbitControls our proxy via `inputElement` instead of passing in the canvas like we do in other non-OffscreenCanvas examples.

Next we can move all the picking event code from the HTML file to the shared three.js code as well while changing `canvas` to `inputElement`.

javascript

```
function getCanvasRelativePosition(event) {
-  const rect = canvas.getBoundingClientRect();
+  const rect = inputElement.getBoundingClientRect();
  return {
    x: event.clientX - rect.left,
    y: event.clientY - rect.top,
  };
}

function setPickPosition(event) {
  const pos = getCanvasRelativePosition(event);
-  sendMouse(
-      (pos.x / canvas.clientWidth ) *  2 - 1,
-      (pos.y / canvas.clientHeight) * -2 + 1);  // note we flip Y
+  pickPosition.x = (pos.x / inputElement.clientWidth ) *  2 - 1;
+  pickPosition.y = (pos.y / inputElement.clientHeight) * -2 + 1;  // note we flip
}

function clearPickPosition() {
  // unlike the mouse which always has a position
  // if the user stops touching the screen we want
  // to stop picking. For now we just pick a value
  // unlikely to pick something
-  sendMouse(-100000, -100000);
+  pickPosition.x = -100000;
+  pickPosition.y = -100000;
}

*inputElement.addEventListener('mousemove', setPickPosition);
*inputElement.addEventListener('mouseout', clearPickPosition);
*inputElement.addEventListener('mouseleave', clearPickPosition);

*inputElement.addEventListener('touchstart', (event) => {
  // prevent the window from scrolling
  event.preventDefault();
  setPickPosition(event.touches[0]);
}, {passive: false});

*inputElement.addEventListener('touchmove', (event) => {
  setPickPosition(event.touches[0]);
});

*inputElement.addEventListener('touchend', clearPickPosition);
```

Back in the main page we need code to send messages for all the events we enumerated above.

javascript

```
let nextProxyId = 0;
class ElementProxy {
  constructor(element, worker, eventHandlers) {
    this.id = nextProxyId++;
    this.worker = worker;
    const sendEvent = (data) => {
      this.worker.postMessage({
        type: 'event',
        id: this.id,
        data,
      });
    };

    // register an id
    worker.postMessage({
      type: 'makeProxy',
      id: this.id,
    });
    for (const [eventName, handler] of Object.entries(eventHandlers)) {
      element.addEventListener(eventName, function(event) {
        handler(event, sendEvent);
      });
    }
  }
}
```

`ElementProxy` takes the element who's events we want to proxy. It then registers an id with the worker by picking one and sending it via the `makeProxy` message we setup earlier. The worker will make an `ElementProxyReceiver` and register it to that id.

We then have an object of event handlers to register. This way we can pass handlers only for these events we want to forward to the worker.

When we start the worker we first make a proxy and pass in our event handlers.

```
javascript
```

```
function startWorker(canvas) {
  const offscreen = canvas.transferControlToOffscreen();
  const worker = new Worker('offscreencanvas-worker-orbitcontrols.js', {type: 'modu

+  const eventHandlers = {
+    contextmenu: preventDefaultHandler,
+    mousedown: mouseEventHandler,
+    mousemove: mouseEventHandler,
+    mouseup: mouseEventHandler,
+    pointerdown: mouseEventHandler,
+    pointermove: mouseEventHandler,
+    pointerup: mouseEventHandler,
+    touchstart: touchEventHandler,
+    touchmove: touchEventHandler,
+    touchend: touchEventHandler,
+    wheel: wheelEventHandler,
+    keydown: filteredKeydownEventHandler,
+  };
+  const proxy = new ElementProxy(canvas, worker, eventHandlers);
  worker.postMessage({
    type: 'start',
    canvas: offscreen,
+    canvasId: proxy.id,
  }, [offscreen]);
  console.log('using OffscreenCanvas');  /* eslint-disable-line no-console */
}
```

And here are the event handlers. All they do is copy a list of properties from the event they receive. They are passed a `sendEvent` function to which they pass the data they make. That function will add the correct id and send it to the worker.

```
javascript
```

```
const mouseEventHandler = makeSendPropertiesHandler([
  'ctrlKey',
  'metaKey',
  'shiftKey',
  'button',
  'pointerType',
  'clientX',
  'clientY',
  'pointerId',
  'pageX',
  'pageY',
]);
const wheelEventHandlerImpl = makeSendPropertiesHandler([
  'deltaX',
  'deltaY',
]);
const keydownEventHandler = makeSendPropertiesHandler([
  'ctrlKey',
  'metaKey',
  'shiftKey',
  'keyCode',
]);

function wheelEventHandler(event, sendFn) {
  event.preventDefault();
  wheelEventHandlerImpl(event, sendFn);
}

function preventDefaultHandler(event) {
  event.preventDefault();
}

function copyProperties(src, properties, dst) {
  for (const name of properties) {
      dst[name] = src[name];
  }
}

function makeSendPropertiesHandler(properties) {
  return function sendProperties(event, sendFn) {
    const data = {type: event.type};
    copyProperties(event, properties, data);
    sendFn(data);
  };
}

function touchEventHandler(event, sendFn) {
  // preventDefault() fixes mousemove, mouseup and mousedown
  // firing when doing a simple touchup touchdown
  // Happens only at offscreen canvas
  event.preventDefault();
  const touches = [];
  const data = {type: event.type, touches};
  for (let i = 0; i < event.touches.length; ++i) {
    const touch = event.touches[i];
    touches.push({
      pageX: touch.pageX,
      pageY: touch.pageY,
      clientX: touch.clientX,
      clientY: touch.clientY,
    });
```

```
  }
  sendFn(data);
}

// The four arrow keys
const orbitKeys = {
  '37': true,  // left
  '38': true,  // up
  '39': true,  // right
  '40': true,  // down
};
function filteredKeydownEventHandler(event, sendFn) {
  const {keyCode} = event;
  if (orbitKeys[keyCode]) {
    event.preventDefault();
    keydownEventHandler(event, sendFn);
  }
}
```

This seems close to running but if we actually try it we'll see that the `OrbitControls` need a few more things.

One is they call `element.focus`. We don't need that to happen in the worker so let's just add a stub.

javascript

```
class ElementProxyReceiver extends THREE.EventDispatcher {
  constructor() {
    super();
  }
  handleEvent(data) {
    this.dispatchEvent(data);
  }
+  focus() {
+    // no-op
+  }
}
```

Another is they call `event.preventDefault` and `event.stopPropagation`. We're already handling that in the main page so those can also be a noop.

javascript

```
+function noop() {
+}

class ElementProxyReceiver extends THREE.EventDispatcher {
  constructor() {
    super();
  }
  handleEvent(data) {
+    data.preventDefault = noop;
+    data.stopPropagation = noop;
    this.dispatchEvent(data);
  }
  focus() {
    // no-op
  }
}
```

Another is they look at `clientWidth` and `clientHeight`. We were passing the size before but we can update the proxy pair to pass that as well.

In the worker...

```
javascript
```

```
class ElementProxyReceiver extends THREE.EventDispatcher {
  constructor() {
    super();
  }
+  get clientWidth() {
+    return this.width;
+  }
+  get clientHeight() {
+    return this.height;
+  }
+  getBoundingClientRect() {
+    return {
+      left: this.left,
+      top: this.top,
+      width: this.width,
+      height: this.height,
+      right: this.left + this.width,
+      bottom: this.top + this.height,
+    };
+  }
  handleEvent(data) {
+    if (data.type === 'size') {
+      this.left = data.left;
+      this.top = data.top;
+      this.width = data.width;
+      this.height = data.height;
+      return;
+    }
    data.preventDefault = noop;
    data.stopPropagation = noop;
    this.dispatchEvent(data);
  }
  focus() {
    // no-op
  }
}
```

back in the main page we need to send the size and the left and top positions as well. Note that as is we don't handle if the canvas moves, only if it resizes. If you wanted to handle moving you'd need to call `sendSize` anytime something moved the canvas.

```
javascript
```

```
class ElementProxy {
  constructor(element, worker, eventHandlers) {
    this.id = nextProxyId++;
    this.worker = worker;
    const sendEvent = (data) => {
      this.worker.postMessage({
        type: 'event',
        id: this.id,
        data,
      });
    };

    // register an id
    worker.postMessage({
      type: 'makeProxy',
      id: this.id,
    });
+    sendSize();
    for (const [eventName, handler] of Object.entries(eventHandlers)) {
      element.addEventListener(eventName, function(event) {
        handler(event, sendEvent);
      });
    }

+    function sendSize() {
+      const rect = element.getBoundingClientRect();
+      sendEvent({
+        type: 'size',
+        left: rect.left,
+        top: rect.top,
+        width: element.clientWidth,
+        height: element.clientHeight,
+      });
+    }
+
+    window.addEventListener('resize', sendSize);
  }
}
```

and in our shared three.js code we no longer need `state`

```
javascript
```

```
-export const state = {
-  width: 300,   // canvas default
-  height: 150,  // canvas default
-};

...

function resizeRendererToDisplaySize(renderer) {
  const canvas = renderer.domElement;
-  const width = state.width;
-  const height = state.height;
+  const width = inputElement.clientWidth;
+  const height = inputElement.clientHeight;
  const needResize = canvas.width !== width || canvas.height !== height;
  if (needResize) {
    renderer.setSize(width, height, false);
  }
  return needResize;
}

function render(time) {
  time *= 0.001;

  if (resizeRendererToDisplaySize(renderer)) {
-    camera.aspect = state.width / state.height;
+    camera.aspect = inputElement.clientWidth / inputElement.clientHeight;
    camera.updateProjectionMatrix();
  }

  ...
```

A few more hacks. The OrbitControls add `pointermove` and `pointerup` events to the `ownerDocument` of the element to handle mouse capture (when the mouse goes outside the window).

Further the code references the global `document` but there is no global document in a worker.

We can solve all of these with a 2 quick hacks. In our worker code we'll re-use our proxy for both problems.

javascript

```
function start(data) {
  const proxy = proxyManager.getProxy(data.canvasId);
+  proxy.ownerDocument = proxy; // HACK!
+  self.document = {} // HACK!
  init({
    canvas: data.canvas,
    inputElement: proxy,
  });
}
```

This will give the `OrbitControls` something to inspect which matches their expectations.

I know that was kind of hard to follow. The short version is: `ElementProxy` runs on the main page and forwards DOM events to `ElementProxyReceiver` in the worker which masquerades as an `HTMLElement` that we can use both with the `OrbitControls` and with our own code.

The final thing is our fallback when we are not using OffscreenCanvas. All we have to do is pass the canvas itself as our `inputElement`.

javascript

```
function startMainPage(canvas) {
-  init({canvas});
+  init({canvas, inputElement: canvas});
  console.log('using regular canvas');
}
```

and now we should have OrbitControls working with OffscreenCanvas

click here to open in a separate window

This is probably the most complicated example on this site. It's a little hard to follow because there are 3 files involved for each sample. The HTML file, the worker file, the shared three.js code.

I hope it wasn't too difficult to understand and that it provided some useful examples of working with three.js, OffscreenCanvas and web workers.

UX & UI Integration

Responsive Design

GUIDE

Source: https://threejs.org/manual/en/responsive.html

This article explains how to make three.js applications responsive to different screen sizes and display situations. It covers CSS-based sizing, handling aspect ratios, managing canvas resolution, and dealing with HD-DPI displays.

Responsive Design

This is the second article in a series of articles about three.js. The first article was about fundamentals. If you haven't read that yet you might want to start there.

This article is about how to make your three.js app be responsive to any situation. Making a webpage responsive generally refers to the page displaying well on different sized displays from desktops to tablets to phones.

For three.js there are even more situations to consider. For example, a 3D editor with controls on the left, right, top, or bottom is something we might want to handle. A live diagram in the middle of a document is another example.

The last sample we had used a plain canvas with no CSS and no size

```html
<canvas id="c"></canvas>
```

Making the Canvas Fill the Page

That canvas defaults to 300x150 CSS pixels in size.

In the web platform the recommended way to set the size of something is to use CSS.

Let's make the canvas fill the page by adding CSS

```css
<style>
html, body {
   margin: 0;
   height: 100%;
}
#c {
   width: 100%;
   height: 100%;
   display: block;
}
</style>
```

Understanding the CSS Setup

In HTML the body has a margin of 5 pixels by default so setting the margin to 0 removes the margin. Setting the html and body height to 100% makes them fill the window. Otherwise they are only as large as the content that fills them.

Next we tell the `id=c` element to be 100% the size of its container which in this case is the body of the document.

Finally we set its `display` mode to `block`. A canvas's default display mode is `inline`. Inline elements can end up adding whitespace to what is displayed. By setting the canvas to `block` that issue goes away.

You can see the canvas is now filling the page but there are 2 problems. One our cubes are stretched. They are not cubes they are more like boxes. Too tall or too wide. Open the example in its own window and resize it. You'll see how the cubes get stretched wide and tall.

The second problem is they look low resolution or blocky and blurry. Stretch the window really large and you'll really see the issue.

Fixing the Stretchy Problem

Let's fix the stretchy problem first. To do that we need to set the aspect of the camera to the aspect of the canvas's display size. We can do that by looking at the canvas's `clientWidth` and `clientHeight` properties.

We'll update our render loop like this

```javascript
function render(time) {
  time *= 0.001;

+  const canvas = renderer.domElement;
+  camera.aspect = canvas.clientWidth / canvas.clientHeight;
+  camera.updateProjectionMatrix();

  ...
```

Verifying the Fix

Now the cubes should stop being distorted.

Open the example in a separate window and resize the window and you should see the cubes are no longer stretched tall or wide. They stay the correct aspect regardless of window size.

Fixing the Blockiness

Canvas elements have 2 sizes. One size is the size the canvas is displayed on the page. That's what we set with CSS. The other size is the number of pixels in the canvas itself. This is no different than an image. For example we might have a 128x64 pixel image and using CSS we might display as 400x200 pixels.

```html
<img src="some128x64image.jpg" style="width:400px; height:200px">
```

Understanding Drawing Buffer Size

A canvas's internal size, its resolution, is often called its drawingbuffer size. In three.js we can set the canvas's drawingbuffer size by calling `renderer.setSize`. What size should we pick? The most obvious answer is "the same size the canvas is displayed". Again, to do that we can look at the canvas's `clientWidth` and `clientHeight` properties.

Let's write a function that checks if the renderer's canvas is not already the size it is being displayed as and if so set its size.

Creating the resizeRendererToDisplaySize Function

Notice we check if the canvas actually needs to be resized. Resizing the canvas is an interesting part of the canvas spec and it's best not to set the same size if it's already the size we want.

Once we know if we need to resize or not we then call `renderer.setSize` and pass in the new width and height. It's important to pass `false` at the end. `renderer.setSize` by default sets the canvas's CSS size but doing so is not what we want. We want the browser to continue to work how it does for all other elements which is to use CSS to determine the display size of the element. We don't want canvases used by three to be different than other elements.

Note that our function returns true if the canvas was resized. We can use this to check if there are other things we should update. Let's modify our render loop to use the new function

```javascript
function resizeRendererToDisplaySize(renderer) {
  const canvas = renderer.domElement;
  const width = canvas.clientWidth;
  const height = canvas.clientHeight;
  const needResize = canvas.width !== width || canvas.height !== height;
  if (needResize) {
    renderer.setSize(width, height, false);
  }
  return needResize;
}
```

Updating the Render Loop

Since the aspect is only going to change if the canvas's display size changed we only set the camera's aspect if `resizeRendererToDisplaySize` returns `true`.

```javascript
function render(time) {
  time *= 0.001;

+  if (resizeRendererToDisplaySize(renderer)) {
+    const canvas = renderer.domElement;
+    camera.aspect = canvas.clientWidth / canvas.clientHeight;
+    camera.updateProjectionMatrix();
+  }

  ...
```

Final Result

It should now render with a resolution that matches the display size of the canvas.

To make the point about letting CSS handle the resizing let's take our code and put it in a separate `.js` file. Here then are a few more examples where we let CSS choose the size and notice we had to change zero code for them to work.

Let's put our cubes in the middle of a paragraph of text.

and here's our same code used in an editor style layout where the control area on the right can be resized.

The important part to notice is no code changed. Only our HTML and CSS changed.

Handling HD-DPI displays

HD-DPI stands for high-density dot per inch displays. That's most Macs nowadays and many Windows machines as well as pretty much all smartphones.

The way this works in the browser is they use CSS pixels to set the sizes which are supposed to be the same regardless of how high res the display is. The browser will just render text with more detail but the same physical size.

There are various ways to handle HD-DPI with three.js.

The first one is just not to do anything special. This is arguably the most common. Rendering 3D graphics takes a lot of GPU processing power. Mobile GPUs have less power than desktops, at least as of 2018, and yet mobile phones often have very high resolution displays. The current top of the line phones have an HD-DPI ratio of 3x meaning for every one pixel from a non-HD-DPI display those phones have 9 pixels. That means they have to do 9x the rendering.

Computing 9x the pixels is a lot of work so if we just leave the code as it is we'll compute 1x the pixels and the browser will just draw it at 3x the size (3x by 3x = 9x pixels).

For any heavy three.js app that's probably what you want otherwise you're likely to get a slow framerate.

That said if you actually do want to render at the resolution of the device there are a couple of ways to do this in three.js.

One is to tell three.js a resolution multiplier using `renderer.setPixelRatio`. You ask the browser what the multiplier is from CSS pixels to device pixels and pass that to three.js

```javascript
renderer.setPixelRatio(window.devicePixelRatio);
```

Why setPixelRatio is Not Recommended

After that any calls to `renderer.setSize` will magically use the size you request multiplied by whatever pixel ratio you passed in. This is strongly NOT RECOMMENDED . See below

The other way is to do it yourself when you resize the canvas.

javascript

```
function resizeRendererToDisplaySize(renderer) {
      const canvas = renderer.domElement;
      const pixelRatio = window.devicePixelRatio;
      const width  = Math.floor( canvas.clientWidth  * pixelRatio );
      const height = Math.floor( canvas.clientHeight * pixelRatio );
      const needResize = canvas.width !== width || canvas.height !== height;
      if (needResize) {
        renderer.setSize(width, height, false);
      }
      return needResize;
    }
```

Why Manual Pixel Ratio is Better

This second way is objectively better. Why? Because it means I get what I ask for. There are many cases when using three.js where we need to know the actual size of the canvas's drawingBuffer. For example when making a post processing filter, or if we are making a shader that accesses `gl_FragCoord`, if we are making a screenshot, or reading pixels for GPU picking, for drawing into a 2D canvas, etc... There are many cases where if we use `setPixelRatio` then our actual size will be different than the size we requested and we'll have to guess when to use the size we asked for and when to use the size three.js is actually using. By doing it ourselves we always know the size being used is the size we requested. There is no special case where magic is happening behind the scenes.

Here's an example using the code above.

It might be hard to see the difference but if you have an HD-DPI display and you compare this sample to those above you should notice the edges are more crisp.

HD-DPI: Limiting maximum drawing buffer size

When using a fractional UI scaling factor on some operating system (eg: 150% on OSX or Linux) the assumption that the real physical resolution equals `width * window.devicePixelRatio` and `height * window.devicePixelRatio` no longer holds. This may lead to excessive GPU load, lower frame rates and high power consumption.

A possible mitigation is to cap the maximum internal resolution (e.g. limit width × height) so the buffer remains within safe bounds.

javascript

```
function resizeRendererToDisplaySize(renderer, maxPixelCount=3840*2160) {
      const canvas = renderer.domElement;
      const pixelRatio = window.devicePixelRatio;
      let width  = Math.floor( canvas.clientWidth  * pixelRatio );
      let height = Math.floor( canvas.clientHeight * pixelRatio );
      const pixelCount = width * height;
      const renderScale = pixelCount > maxPixelCount ? Math.sqrt(maxPixelCount / pi
      width = Math.floor(width * renderScale);
      height = Math.floor(height * renderScale);

      const needResize = canvas.width !== width || canvas.height !== height;
      if (needResize) {
        renderer.setSize(width, height, false);
      }
      return needResize;
    }
```

Conclusion

This article covered a very basic but fundamental topic. Next up lets quickly go over the basic primitives that three.js provides.

Multiple Canvases Multiple Scenes

GUIDE

Source: https://threejs.org/manual/en/multiple-scenes.html

A guide explaining how to use THREE.js with multiple virtual scenes rendered into a single canvas, addressing the browser's WebGL context limit and resource sharing issues. It covers the scissor test approach, syncing with scroll, making the system generic, using HTML dataset, adding controls, and an alternative using off-screen canvas copying.

Multiple Canvases Multiple Scenes

A common question is how to use THREE.js with multiple canvases. Let's say you want to make an e-commerce site or you want to make a page with lots of 3D diagrams. At first glance it appears easy. Just make a canvas every where you want a diagram. For each canvas make a Renderer. You'll quickly find though that you run into problems.

The browser limits how many WebGL contexts you can have. Typically that limit is around 8 of them. As soon as you create the 9th context the oldest one will be lost.

WebGL resources can not be shared across contexts. That means if you want to load a 10 meg model into 2 canvases and that model uses 20 meg of textures your 10 meg model will have to be loaded twice and your textures will also be loaded twice.

Nothing can be shared across contexts. This also means things have to be initialized twice, shaders compiled twice, etc. It gets worse as there are more canvases.

So what's the solution? The solution is one canvas that fills the viewport in the background and some other element to represent each "virtual" canvas. We make a single Renderer and then one Scene for each virtual canvas. We'll then check the positions of the virtual canvas elements and if they are on the screen we'll tell THREE.js to draw their scene at the correct place.

With this solution there is only 1 canvas so we solve both problem 1 and 2 above. We won't run into the WebGL context limit because we will only be using one context. We also won't run into the sharing issues for the same reasons.

Let's start with a simple example with just 2 scenes. First we'll make the HTML

Then we can setup the CSS maybe something like this

We set the canvas to fill the screen and we set its z-index to -1 to make it appear behind other elements. We also need to specify some kind of width and height for our virtual canvas elements since there is nothing inside to give them any size.

Now we'll make 2 scenes each with a light and a camera. To one scene we'll add a cube and to another a diamond.

And then we'll make a function to render each scene only if the element is on the screen. We can tell THREE.js to only render to part of the canvas by turning on the scissor test with Renderer.setScissorTest and then setting both the scissor and the viewport with Renderer.setViewport and Renderer.setScissor.

And then our render function will just first clear the screen and then render each scene.

And here it is

You can see where the first is there's a red cube and where the second span is there's a blue diamond.

```
html
```

```
<canvas id="c"></canvas>
<p>
  <span id="box" class="diagram left"></span>
  I love boxes. Presents come in boxes.
  When I find a new box I'm always excited to find out what's inside.
</p>
<p>
  <span id="pyramid" class="diagram right"></span>
  When I was a kid I dreamed of going on an expedition inside a pyramid
  and finding a undiscovered tomb full of mummies and treasure.
</p>
```

css

```
#c {
  position: fixed;
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  display: block;
  z-index: -1;
}
.diagram {
  display: inline-block;
  width: 5em;
  height: 3em;
  border: 1px solid black;
}
.left {
  float: left;
  margin-right: .25em;
}
.right {
  float: right;
  margin-left: .25em;
}
```

javascript

```
function makeScene(elem) {
  const scene = new THREE.Scene();

  const fov = 45;
  const aspect = 2;  // the canvas default
  const near = 0.1;
  const far = 5;
  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
  camera.position.z = 2;
  camera.position.set(0, 1, 2);
  camera.lookAt(0, 0, 0);

  {
    const color = 0xFFFFFF;
    const intensity = 1;
    const light = new THREE.DirectionalLight(color, intensity);
    light.position.set(-1, 2, 4);
    scene.add(light);
  }

  return {scene, camera, elem};
}

function setupScene1() {
  const sceneInfo = makeScene(document.querySelector('#box'));
  const geometry = new THREE.BoxGeometry(1, 1, 1);
  const material = new THREE.MeshPhongMaterial({color: 'red'});
  const mesh = new THREE.Mesh(geometry, material);
  sceneInfo.scene.add(mesh);
  sceneInfo.mesh = mesh;
  return sceneInfo;
}

function setupScene2() {
  const sceneInfo = makeScene(document.querySelector('#pyramid'));
  const radius = .8;
  const widthSegments = 4;
  const heightSegments = 2;
  const geometry = new THREE.SphereGeometry(radius, widthSegments, heightSegments);
  const material = new THREE.MeshPhongMaterial({
    color: 'blue',
    flatShading: true,
  });
  const mesh = new THREE.Mesh(geometry, material);
  sceneInfo.scene.add(mesh);
  sceneInfo.mesh = mesh;
  return sceneInfo;
}

const sceneInfo1 = setupScene1();
const sceneInfo2 = setupScene2();
```

javascript

```
function renderSceneInfo(sceneInfo) {
  const {scene, camera, elem} = sceneInfo;

  // get the viewport relative position of this element
  const {left, right, top, bottom, width, height} =
      elem.getBoundingClientRect();

  const isOffscreen =
      bottom < 0 ||
      top > renderer.domElement.clientHeight ||
      right < 0 ||
      left > renderer.domElement.clientWidth;

  if (isOffscreen) {
    return;
  }

  camera.aspect = width / height;
  camera.updateProjectionMatrix();

  const positiveYUpBottom = canvasRect.height - bottom;
  renderer.setScissor(left, positiveYUpBottom, width, height);
  renderer.setViewport(left, positiveYUpBottom, width, height);

  renderer.render(scene, camera);
}
```

javascript

```
function render(time) {
  time *= 0.001;

  resizeRendererToDisplaySize(renderer);

  renderer.setScissorTest(false);
  renderer.clear(true, true);
  renderer.setScissorTest(true);

  sceneInfo1.mesh.rotation.y = time * .1;
  sceneInfo2.mesh.rotation.y = time * .1;

  renderSceneInfo(sceneInfo1);
  renderSceneInfo(sceneInfo2);

  requestAnimationFrame(render);
}
```

Syncing up

The code above works but there is one minor issue. If your scenes are complicated or if for whatever reason it takes too long to render, the position of the scenes drawn into the canvas will lag behind the rest of the page.

If we give each area a border

And we set the background of each scene

And if we quickly scroll up and down we'll see the issue. Here's an animation of scrolling slowed down by 10x.

We can switch to a different method which has a different tradeoff. We'll switch the canvas's CSS from position: fixed to position: absolute.

Then we'll set the canvas's transform to move it so the top of the canvas is at the top of whatever part the page is currently scrolled to.

position: fixed kept the canvas from scrolling at all while the rest of the page scrolled over it. position: absolute will let the canvas scroll with the rest of the page which means whatever we draw will stick with the page as it scrolls even if we're too slow to render. When we finally get a chance to render then we move the canvas so it matches where the page has been scrolled and then we re-render. This means only the edges of the window will show some un-rendered bits for a moment but the stuff in the middle of the page should match up and not slide. Here's a view of the results of the new method slowed down 10x.

```
css
```

```
 .diagram {
   display: inline-block;
   width: 5em;
   height: 3em;
+  border: 1px solid black;
 }
```

```
javascript
```

```
const scene = new THREE.Scene();
+scene.background = new THREE.Color('red');
```

```
css
```

```
 #c {
-  position: fixed;
+  position: absolute;
```

```
javascript
```

```
function render(time) {
  ...

  const transform = translateY(${window.scrollY}px);
  renderer.domElement.style.transform = transform;
```

Making it more Generic

Now that we've gotten multiple scenes working let's make this just slightly more generic.

We could make it so the main render function, the one managing the canvas, just has a list of elements and their associated render function. For each element it would check if the element is on screen and if so call the corresponding render function. In this way we'd have a generic system where individual scenes aren't really aware they are being rendered in some smaller space.

Here's the main render function

You can see it loops over sceneElements which it expects is an array of objects each of which have an elem and fn property.

It checks if the element is on screen. If it is it calls fn and passes it the current time and its rectangle.

Now the setup code for each scene just adds itself to the list of scenes

With that we no longer need sceneInfo1 and sceneInfo2 and the code that was rotating the meshes is now specific to each scene.

```
javascript
```

```javascript
const sceneElements = [];
function addScene(elem, fn) {
  sceneElements.push({elem, fn});
}

function render(time) {
  time *= 0.001;

  resizeRendererToDisplaySize(renderer);

  renderer.setScissorTest(false);
  renderer.setClearColor(clearColor, 0);
  renderer.clear(true, true);
  renderer.setScissorTest(true);

  const transform = translateY(${window.scrollY}px);
  renderer.domElement.style.transform = transform;

  for (const {elem, fn} of sceneElements) {
    // get the viewport relative position of this element
    const rect = elem.getBoundingClientRect();
    const {left, right, top, bottom, width, height} = rect;

    const isOffscreen =
        bottom < 0 ||
        top > renderer.domElement.clientHeight ||
        right < 0 ||
        left > renderer.domElement.clientWidth;

    if (!isOffscreen) {
      const positiveYUpBottom = renderer.domElement.clientHeight - bottom;
      renderer.setScissor(left, positiveYUpBottom, width, height);
      renderer.setViewport(left, positiveYUpBottom, width, height);

      fn(time, rect);
    }
  }

  requestAnimationFrame(render);
}
```

javascript

```
{
  const elem = document.querySelector('#box');
  const {scene, camera} = makeScene();
  const geometry = new THREE.BoxGeometry(1, 1, 1);
  const material = new THREE.MeshPhongMaterial({color: 'red'});
  const mesh = new THREE.Mesh(geometry, material);
  scene.add(mesh);
  addScene(elem, (time, rect) => {
    camera.aspect = rect.width / rect.height;
    camera.updateProjectionMatrix();
    mesh.rotation.y = time * .1;
    renderer.render(scene, camera);
  });
}

{
  const elem = document.querySelector('#pyramid');
  const {scene, camera} = makeScene();
  const radius = .8;
  const widthSegments = 4;
  const heightSegments = 2;
  const geometry = new THREE.SphereGeometry(radius, widthSegments, heightSegments);
  const material = new THREE.MeshPhongMaterial({
    color: 'blue',
    flatShading: true,
  });
  const mesh = new THREE.Mesh(geometry, material);
  scene.add(mesh);
  addScene(elem, (time, rect) => {
    camera.aspect = rect.width / rect.height;
    camera.updateProjectionMatrix();
    mesh.rotation.y = time * .1;
    renderer.render(scene, camera);
  });
}
```

Using HTML Dataset

One last even more generic thing we can do is use HTML dataset. This is a way to add your own data to an HTML element. Instead of using id="..." we'll use data-diagram="..." like this

We can them change the CSS selector to select for that

We'll change the scene setup code to just be a map of names to scene initialization functions that return a scene render function.

And to init we can just use querySelectorAll to find all the diagrams and call the corresponding init function for that diagram.

No change to the visuals but the code is even more generic.

html

```
<canvas id="c"></canvas>
<p>
-  <span id="box" class="diagram left"></span>
+  <span data-diagram="box" class="left"></span>
  I love boxes. Presents come in boxes.
  When I find a new box I'm always excited to find out what's inside.
</p>
<p>
-  <span id="pyramid" class="diagram left"></span>
+  <span data-diagram="pyramid" class="right"></span>
  When I was a kid I dreamed of going on an expedition inside a pyramid
  and finding a undiscovered tomb full of mummies and treasure.
</p>
```

css

```
-.diagram
+*[data-diagram] {
  display: inline-block;
  width: 5em;
  height: 3em;
}
```

javascript

```
const sceneInitFunctionsByName = {
  'box': () => {
    const {scene, camera} = makeScene();
    const geometry = new THREE.BoxGeometry(1, 1, 1);
    const material = new THREE.MeshPhongMaterial({color: 'red'});
    const mesh = new THREE.Mesh(geometry, material);
    scene.add(mesh);
    return (time, rect) => {
      mesh.rotation.y = time * .1;
      camera.aspect = rect.width / rect.height;
      camera.updateProjectionMatrix();
      renderer.render(scene, camera);
    };
  },
  'pyramid': () => {
    const {scene, camera} = makeScene();
    const radius = .8;
    const widthSegments = 4;
    const heightSegments = 2;
    const geometry = new THREE.SphereGeometry(radius, widthSegments, heightSegments
    const material = new THREE.MeshPhongMaterial({
      color: 'blue',
      flatShading: true,
    });
    const mesh = new THREE.Mesh(geometry, material);
    scene.add(mesh);
    return (time, rect) => {
      mesh.rotation.y = time * .1;
      camera.aspect = rect.width / rect.height;
      camera.updateProjectionMatrix();
      renderer.render(scene, camera);
    };
  },
};
```

javascript

```
document.querySelectorAll('[data-diagram]').forEach((elem) => {
  const sceneName = elem.dataset.diagram;
  const sceneInitFunction = sceneInitFunctionsByName[sceneName];
  const sceneRenderFunction = sceneInitFunction(elem);
  addScene(elem, sceneRenderFunction);
});
```

Adding Controls to each element

Adding interactively, for example a TrackballControls is just as easy. First we add the script for the control.

And then we can add a TrackballControls to each scene passing in the element associated with that scene.

You'll notice we added the camera to the scene and the light to the camera. This makes the light relative to the camera. Since the TrackballControls are moving the camera this is probably what we want. It keeps the light shining on the side of the object we are looking at.

We need up update those controls in our render functions

And now if you drag the objects they'll rotate.

These techniques are used on this site itself. In particular the article about primitives and the article about materials use this technique to add the various examples throughout the article.

One more solution would be to render to an off screen canvas and copy the result to a 2D canvas at each element. The advantage to this solution is there is no limit on how you can composite each separate area. With the previous solution we and a single canvas in the background. With this solution we have normal HTML elements.

The disadvantage is it's slower because a copy has to happen for each area. How much slower depends on the browser and the GPU.

The changes needed are pretty small

First we'll change HTML as we no longer need a canvas in the page

then we'll change the CSS

We've made all canvases fill their container.

Now let's change the JavaScript. First we no longer look up the canvas. Instead we create one. We also just turn on the scissor test at the beginning.

Then for each scene we create a 2D rendering context and append its canvas to the element for that scene

Then when rendering, if the renderer's canvas is not big enough to render this area we increase its size. As well if this area's canvas is the wrong size we change its size. Finally we set the scissor and viewport, render the scene for this area, then copy the result to the area's canvas.

The result looks the same

One other advantage to this solution is you could potentially use OffscreenCanvas to render from a web worker and still use this technique.

javascript

```
import {TrackballControls} from 'three/addons/controls/TrackballControls.js';
```

javascript

```
function makeScene(elem) {
  const scene = new THREE.Scene();

  const fov = 45;
  const aspect = 2;  // the canvas default
  const near = 0.1;
  const far = 5;
  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);
  camera.position.set(0, 1, 2);
  camera.lookAt(0, 0, 0);
  scene.add(camera);

  const controls = new TrackballControls(camera, elem);
  controls.noZoom = true;
  controls.noPan = true;

  {
    const color = 0xFFFFFF;
    const intensity = 1;
    const light = new THREE.DirectionalLight(color, intensity);
    light.position.set(-1, 2, 4);
    camera.add(light);
  }

  return {scene, camera, controls};
}
```

javascript

```
const sceneInitFunctionsByName = {
 'box': (elem) => {
    const {scene, camera, controls} = makeScene(elem);
    const geometry = new THREE.BoxGeometry(1, 1, 1);
    const material = new THREE.MeshPhongMaterial({color: 'red'});
    const mesh = new THREE.Mesh(geometry, material);
    scene.add(mesh);
    return (time, rect) => {
      mesh.rotation.y = time * .1;
      camera.aspect = rect.width / rect.height;
      camera.updateProjectionMatrix();
      controls.handleResize();
      controls.update();
      renderer.render(scene, camera);
    };
  },
  'pyramid': (elem) => {
    const {scene, camera, controls} = makeScene(elem);
    const radius = .8;
    const widthSegments = 4;
    const heightSegments = 2;
    const geometry = new THREE.SphereGeometry(radius, widthSegments, heightSegments
    const material = new THREE.MeshPhongMaterial({
      color: 'blue',
      flatShading: true,
    });
    const mesh = new THREE.Mesh(geometry, material);
    scene.add(mesh);
    return (time, rect) => {
      mesh.rotation.y = time * .1;
      camera.aspect = rect.width / rect.height;
      camera.updateProjectionMatrix();
      controls.handleResize();
      controls.update();
      renderer.render(scene, camera);
    };
  },
};
```

html

```
<body>
-  <canvas id="c"></canvas>
  ...
</body>
```

css

```
canvas {
  width: 100%;
  height: 100%;
  display: block;
}
*[data-diagram] {
  display: inline-block;
  width: 5em;
  height: 3em;
}
```

javascript

```
function main() {
  const canvas = document.createElement('canvas');
  const renderer = new THREE.WebGLRenderer({antialias: true, canvas, alpha: true});
  renderer.setScissorTest(true);

  ...
```

javascript

```
const sceneElements = [];
function addScene(elem, fn) {
  const ctx = document.createElement('canvas').getContext('2d');
  elem.appendChild(ctx.canvas);
  sceneElements.push({elem, ctx, fn});
}
```

javascript

```
function render(time) {
  time *= 0.001;

  for (const {elem, fn, ctx} of sceneElements) {
    // get the viewport relative position of this element
    const rect = elem.getBoundingClientRect();
    const {left, right, top, bottom, width, height} = rect;
    const rendererCanvas = renderer.domElement;

    const isOffscreen =
        bottom < 0 ||
        top > window.innerHeight ||
        right < 0 ||
        left > window.innerWidth;

    if (!isOffscreen) {
      // make sure the renderer's canvas is big enough
      if (rendererCanvas.width < width || rendererCanvas.height < height) {
        renderer.setSize(width, height, false);
      }

      // make sure the canvas for this area is the same size as the area
      if (ctx.canvas.width !== width || ctx.canvas.height !== height) {
        ctx.canvas.width = width;
        ctx.canvas.height = height;
      }

      renderer.setScissor(0, 0, width, height);
      renderer.setViewport(0, 0, width, height);

      fn(time, rect);

      // copy the rendered scene to this element's canvas
      ctx.globalCompositeOperation = 'copy';
      ctx.drawImage(
          rendererCanvas,
          0, rendererCanvas.height - height, width, height,  // src rect
          0, 0, width, height);                              // dst rect
    }
  }

  requestAnimationFrame(render);
}
```

Billboards

GUIDE

Source: https://threejs.org/manual/en/billboards.html

A guide on using Three.js Sprite and SpriteMaterial to create billboards — 2D planes that always face the camera — for purposes like character labels and rendering facades of 3D objects to improve performance.

Billboards

In a previous article we used a CanvasTexture to make labels / badges on characters. Sometimes we'd like to make labels or other things that always face the camera. Three.js provides the Sprite and SpriteMaterial to make this happen.

Let's change the badge example from the article on canvas textures to use Sprite and SpriteMaterial

javascript

```
function makePerson(x, labelWidth, size, name, color) {
  const canvas = makeLabelCanvas(labelWidth, size, name);
  const texture = new THREE.CanvasTexture(canvas);
  // because our canvas is likely not a power of 2
  // in both dimensions set the filtering appropriately.
  texture.minFilter = THREE.LinearFilter;
  texture.wrapS = THREE.ClampToEdgeWrapping;
  texture.wrapT = THREE.ClampToEdgeWrapping;

-  const labelMaterial = new THREE.MeshBasicMaterial({
+  const labelMaterial = new THREE.SpriteMaterial({
    map: texture,
-    side: THREE.DoubleSide,
    transparent: true,
  });

  const root = new THREE.Object3D();
  root.position.x = x;

  const body = new THREE.Mesh(bodyGeometry, bodyMaterial);
  root.add(body);
  body.position.y = bodyHeight / 2;

  const head = new THREE.Mesh(headGeometry, bodyMaterial);
  root.add(head);
  head.position.y = bodyHeight + headRadius * 1.1;

-  const label = new THREE.Mesh(labelGeometry, labelMaterial);
+  const label = new THREE.Sprite(labelMaterial);
  root.add(label);
  label.position.y = bodyHeight * 4 / 5;
  label.position.z = bodyRadiusTop * 1.01;
```

Fixing label intersection

and the labels now always face the camera

[click here to open in a separate window]

One problem is from certain angles the labels now intersect the characters.

We can move the position of the labels to fix.

```javascript
+// if units are meters then 0.01 here makes size
+// of the label into centimeters.
+const labelBaseScale = 0.01;
const label = new THREE.Sprite(labelMaterial);
root.add(label);
-label.position.y = bodyHeight * 4 / 5;
-label.position.z = bodyRadiusTop * 1.01;
+label.position.y = head.position.y + headRadius + size * labelBaseScale;

-// if units are meters then 0.01 here makes size
-// of the label into centimeters.
-const labelBaseScale = 0.01;
label.scale.x = canvas.width  * labelBaseScale;
label.scale.y = canvas.height * labelBaseScale;
```

Facades with billboards

Another thing we can do with billboards is draw facades.

Instead of drawing 3D objects we draw 2D planes with an image of 3D objects. This is often faster than drawing 3D objects.

For example let's make a scene with grid of trees. We'll make each tree from a cylinder for the base and a cone for the top.

First we make the cone and cylinder geometry and materials that all the trees will share

```javascript
const trunkRadius = .2;
const trunkHeight = 1;
const trunkRadialSegments = 12;
const trunkGeometry = new THREE.CylinderGeometry(
    trunkRadius, trunkRadius, trunkHeight, trunkRadialSegments);

const topRadius = trunkRadius * 4;
const topHeight = trunkHeight * 2;
const topSegments = 12;
const topGeometry = new THREE.ConeGeometry(
    topRadius, topHeight, topSegments);

const trunkMaterial = new THREE.MeshPhongMaterial({color: 'brown'});
const topMaterial = new THREE.MeshPhongMaterial({color: 'green'});
```

Making tree meshes

Then we'll make a function that makes a Mesh each for the trunk and top of a tree and parents both to an Object3D.

```javascript
function makeTree(x, z) {
  const root = new THREE.Object3D();
  const trunk = new THREE.Mesh(trunkGeometry, trunkMaterial);
  trunk.position.y = trunkHeight / 2;
  root.add(trunk);

  const top = new THREE.Mesh(topGeometry, topMaterial);
  top.position.y = trunkHeight + topHeight / 2;
  root.add(top);

  root.position.set(x, 0, z);
  scene.add(root);

  return root;
}
```

Placing trees in a grid

Then we'll make a loop to place a grid of trees.

```javascript
for (let z = -50; z <= 50; z += 10) {
  for (let x = -50; x <= 50; x += 10) {
    makeTree(x, z);
  }
}
```

Adding ground and background

Let's also add a ground plane while we're at it

```javascript
// add ground
{
  const size = 400;
  const geometry = new THREE.PlaneGeometry(size, size);
  const material = new THREE.MeshPhongMaterial({color: 'gray'});
  const mesh = new THREE.Mesh(geometry, material);
  mesh.rotation.x = Math.PI * -0.5;
  scene.add(mesh);
}
```

Setting the scene background

and change the background to light blue

```javascript
 const scene = new THREE.Scene();
 -scene.background = new THREE.Color('white');
 +scene.background = new THREE.Color('lightblue');
```

Rendering an object to a texture

and we get a grid of trees

[click here to open in a separate window]

There are 11x11 or 121 trees. Each tree is made from a 12 polygon cone and a 48 polygon trunk so each tree is 60 polygons. 121 * 60 is 7260 polygons. That's not that many but of course a more detailed 3D tree might be 1000-3000 polygons. If they were 3000 polygons each then 121 trees would be 363000 polygons to draw.

Using facades we can bring that number down.

We could manually create a facade in some painting program but let's write some code to try to generate one.

Let's write some code to render an object to a texture using a RenderTarget. We covered rendering to a RenderTarget in the article on render targets.

```javascript
```

```
function frameArea(sizeToFitOnScreen, boxSize, boxCenter, camera) {
  const halfSizeToFitOnScreen = sizeToFitOnScreen * 0.5;
  const halfFovY = THREE.MathUtils.degToRad(camera.fov * .5);
  const distance = halfSizeToFitOnScreen / Math.tan(halfFovY);

  camera.position.copy(boxCenter);
  camera.position.z += distance;

  // pick some near and far values for the frustum that
  // will contain the box.
  camera.near = boxSize / 100;
  camera.far = boxSize * 100;

  camera.updateProjectionMatrix();
}

function makeSpriteTexture(textureSize, obj) {
  const rt = new THREE.WebGLRenderTarget(textureSize, textureSize);

  const aspect = 1;  // because the render target is square
  const camera = new THREE.PerspectiveCamera(fov, aspect, near, far);

  scene.add(obj);

  // compute the box that contains obj
  const box = new THREE.Box3().setFromObject(obj);

  const boxSize = box.getSize(new THREE.Vector3());
  const boxCenter = box.getCenter(new THREE.Vector3());

  // set the camera to frame the box
  const fudge = 1.1;
  const size = Math.max(...boxSize.toArray()) * fudge;
  frameArea(size, size, boxCenter, camera);

  renderer.autoClear = false;
  renderer.setRenderTarget(rt);
  renderer.render(scene, camera);
  renderer.setRenderTarget(null);
  renderer.autoClear = true;

  scene.remove(obj);

  return {
    position: boxCenter.multiplyScalar(fudge),
    scale: size,
    texture: rt.texture,
  };
}
```

Notes on the facade rendering code

Some things to note about the code above:

We're using the field of view (fov) defined above this code.

We're computing a box that contains the tree the same way we did in the article on loading a .obj file with a few minor changes.

We call frameArea again adapted the article on loading a .obj file. In this case we compute how far the camera needs to be away from the object given its field of view to contain the object. We then position the camera -z that distance from the center of the box that contains the object.

We multiply the size we want to fit by 1.1 (fudge) to make sure the tree fits completely in the render target. The issue here is the size we're using to calculate if the object fits in the camera's view is not taking into account that the very edges of the object will end up dipping outside area we calculated. We could compute how to make 100% of the box fit but that would waste space as well so instead we just *fudge* it.

Then we render to the render target and remove the object from the scene.

It's important to note we need the lights in the scene but we need to make sure nothing else is in the scene.

We also need to not set a background color on the scene

```javascript
const scene = new THREE.Scene();
-scene.background = new THREE.Color('lightblue');
```

Making facades for a grid of trees

Finally we've made the texture we return it and the position and scale we need to make the facade so that it will appear to be in the same place.

We then make a tree and call this code and pass it in

```javascript
// make billboard texture
const tree = makeTree(0, 0);
const facadeSize = 64;
const treeSpriteInfo = makeSpriteTexture(facadeSize, tree);
```

Replacing tree models with sprites

We can then make a grid of facades instead of a grid of tree models

```javascript
+function makeSprite(spriteInfo, x, z) {
+  const {texture, offset, scale} = spriteInfo;
+  const mat = new THREE.SpriteMaterial({
+    map: texture,
+    transparent: true,
+  });
+  const sprite = new THREE.Sprite(mat);
+  scene.add(sprite);
+  sprite.position.set(
+      offset.x + x,
+      offset.y,
+      offset.z + z);
+  sprite.scale.set(scale, scale, scale);
+}

for (let z = -50; z <= 50; z += 10) {
  for (let x = -50; x <= 50; x += 10) {
-    makeTree(x, z);
+    makeSprite(treeSpriteInfo, x, z);
  }
}
```

Restoring the background

In the code above we apply the offset and scale needed to position the facade so it appears the same place the original tree would have appeared.

Now that we're done making the tree facade texture we can set the background again

```javascript
scene.background = new THREE.Color('lightblue');
```

Conclusion

and now we get a scene of tree facades

[click here to open in a separate window]

Compare to the trees models above and you can see it looks fairly similar. We used a low-res texture, just 64x64 pixels so the facades are blocky. You could increase the resolution. Often facades are used only in the far distance when they are fairly small so a low-res texture is enough and it saves on drawing detailed trees that are only a few pixels big when far away.

Another issue is we are only viewing the tree from one side. This is often solved by rendering more facades, say from 8 directions around the object and then setting which facade to show based on which direction the camera is looking at the facade.

Whether or not you use facades is up to you but hopefully this article gave you some ideas and suggested some solutions if you decide to use them.

Maintenance & Debugging

How to dispose of Objects

GUIDE

Source: https://threejs.org/manual/en/how-to-dispose-of-objects.html

A guide covering how to properly dispose of three.js objects such as geometries, materials, textures, render targets, and skinned meshes to free WebGL-related resources and avoid memory leaks.

How to dispose of Objects

One important aspect in order to improve performance and avoid memory leaks in your application is the disposal of unused library entities. Whenever you create an instance of a *three.js* type, you allocate a certain amount of memory. However, *three.js* creates for specific objects like geometries or materials WebGL related entities like buffers or shader programs which are necessary for rendering. It's important to highlight that these objects are not released automatically. Instead, the application has to use a special API in order to free such resources. This guide provides a brief overview about how this API is used and what objects are relevant in this context.

Geometries

A geometry usually represents vertex information defined as a collection of attributes. *three.js* internally creates an object of type WebGLBuffer for each attribute. These entities are only deleted if you call `BufferGeometry.dispose()`. If a geometry becomes obsolete in your application, execute the method to free all related resources.

Materials

A material defines how objects are rendered. *three.js* uses the information of a material definition in order to construct a shader program for rendering. Shader programs can only be deleted if the respective material is disposed. For performance reasons, *three.js* tries to reuse existing shader programs if possible. So a shader program is only deleted if all related materials are disposed. You can indicate the disposal of a material by executing `Material.dispose()`.

Textures

The disposal of a material has no effect on textures. They are handled separately since a single texture can be used by multiple materials at the same time. Whenever you create an instance of `Texture`, three.js internally creates an instance of WebGLTexture. Similar to buffers, this object can only be deleted by calling `Texture.dispose()`.

If you use an `ImageBitmap` as the texture's data source, you have to call ImageBitmap.close() at the application level to dispose of all CPU-side resources. An automated call of `ImageBitmap.close()` in `Texture.dispose()` is not possible, since the image bitmap becomes unusable, and the engine has no way of knowing if the image bitmap is used elsewhere.

Render Targets

Objects of type `WebGLRenderTarget` not only allocate an instance of WebGLTexture but also WebGLFramebuffers and WebGLRenderbuffers for realizing custom rendering destinations. These objects are only deallocated by executing `WebGLRenderTarget.dispose()`.

Skinned Mesh

Skinned meshes represent their bone hierarchy as skeletons. If you don't need a skinned mesh anymore, consider to call `Skeleton.dispose()` on the skeleton to free internal resources. Keep in mind that skeletons can be shared across multiple skinned meshes, so only call `dispose()` if the skeleton is not used by other active skinned meshes.

Miscellaneous

There are other classes from the examples directory like controls or post processing passes which provide `dispose()` methods in order to remove internal event listeners or render targets. In general, it's recommended to check the API or documentation of a class and watch for `dispose()`. If present, you should use it when cleaning things up.

Why can't *three.js* dispose objects automatically?

This question was asked many times by the community so it's important to clarify this matter. Fact is that *three.js* does not know the lifetime or scope of user-created entities like geometries or materials. This is the responsibility of the application. For example even if a material is currently not used for rendering, it might be necessary for the next frame. So if the application decides that a certain object can be deleted, it has to notify the engine via calling the respective `dispose()` method.

Does removing a mesh from the scene also dispose its geometry and material?

No, you have to explicitly dispose the geometry and material via *dispose()*. Keep in mind that geometries and materials can be shared among 3D objects like meshes.

Does *three.js* provide information about the amount of cached objects?

Yes. It's possible to evaluate `renderer.info`, a special property of the renderer with a series of statistical information about the graphics board memory and the rendering process. Among other things, it tells you how many textures, geometries and shader programs are internally stored. If you notice performance problems in your application, it's a good idea to debug this property in order to easily identify a memory leak.

What happens when you call `dispose()` on a texture but the image is not loaded yet?

Internal resources for a texture are only allocated if the image has fully loaded. If you dispose a texture before the image was loaded, nothing happens. No resources were allocated so there is also no need for clean up.

What happens when I call `dispose()` and then use the respective object at a later point?

That depends. For geometries, materials, textures, render targets and post processing passes the deleted internal resources can be created again by the engine. So no runtime error will occur but you might notice a negative performance impact for the current frame, especially when shader programs have to be compiled.

Controls and renderers are an exception. Instances of these classes can not be used after `dispose()` has been called. You have to create new instances in this case.

How should I manage *three.js* objects in my app? When do I know how to dispose things?

In general, there is no definite recommendation for this. It highly depends on the specific use case when calling `dispose()` is appropriate. It's important to highlight that it's not always necessary to dispose objects all the time. A good example for this is a game which consists of multiple levels. A good place for object disposal is when switching the level. The app could traverse through the old scene and dispose all obsolete materials, geometries and textures. As mentioned in the previous section, it does not produce a runtime error if you dispose an object that is actually still in use. The worst thing that can happen is performance drop for a single frame.

Why `renderer.info.memory` is still reporting geometries and textures after traversing the scene and disposing all reachable textures and geometries?

In certain cases, there are some textures and geometries used internally by Three.js that are not reachable when traversing the scene graph in order to be disposed. It is expected that `renderer.info.memory` will still report them even after a full scene cleanup. However, they do not leak, but they are reused on consecutive scene cleanup/repopulating cycles.

These cases could be related to using `material.envMap`, `scene.background`, `scene.environment`, or other contexts that would require the engine to create textures or geometries for internal use.

Examples that demonstrate the usage of dispose()

WebGL / test / memory

WebGL / test / memory2

How to update Things

GUIDE

Source: https://threejs.org/manual/en/how-to-update-things.html

A guide explaining how to update various Three.js objects including matrix transforms, BufferGeometries, Materials, Textures, Cameras, InstancedMesh, and SkinnedMesh.

How to update Things

All objects by default automatically update their matrices if they have been added to the scene with

```
const object = new THREE.Object3D();
scene.add( object );
```

or if they are the child of another object that has been added to the scene:

```
const object1 = new THREE.Object3D();
const object2 = new THREE.Object3D();

object1.add( object2 );
scene.add( object1 ); //object1 and object2 will automatically update their matrice
```

However, if you know the object will be static, you can disable this and update the transform matrix manually just when needed.

```
object.matrixAutoUpdate = false;
object.updateMatrix();
```

javascript

```
const object = new THREE.Object3D();
scene.add( object );
```

javascript

```
const object1 = new THREE.Object3D();
const object2 = new THREE.Object3D();

object1.add( object2 );
scene.add( object1 ); //object1 and object2 will automatically update their matrice
```

javascript

```
object.matrixAutoUpdate = false;
object.updateMatrix();
```

BufferGeometry

BufferGeometries store information (such as vertex positions, face indices, normals, colors, UVs, and any custom attributes) in attribute buffers - that is, typed arrays. This makes them generally faster than standard Geometries, at the cost of being somewhat harder to work with.

With regards to updating BufferGeometries, the most important thing to understand is that you cannot resize buffers (this is very costly, basically the equivalent to creating a new geometry). You can however update the content of buffers.

This means that if you know an attribute of your BufferGeometry will grow, say the number of vertices, you must pre-allocate a buffer large enough to hold any new vertices that may be created. Of course, this also means that there will be a maximum size for your BufferGeometry - there is no way to create a BufferGeometry that can efficiently be extended indefinitely.

We'll use the example of a line that gets extended at render time. We'll allocate space in the buffer for 500 vertices but draw only two at first, using `BufferGeometry.drawRange`.

Next we'll randomly add points to the line using a pattern like:

If you want to change the number of points rendered after the first render, do this:

If you want to change the position data values after the first render, you need to set the needsUpdate flag like so:

If you change the position data values after the initial render, you may need to recompute bounding volumes so other features of the engine like view frustum culling or helpers properly work.

Here is a fiddle showing an animated line which you can adapt to your use case.

javascript

```javascript
const MAX_POINTS = 500;

// geometry
const geometry = new THREE.BufferGeometry();

// attributes
const positions = new Float32Array( MAX_POINTS * 3 ); // 3 floats (x, y and z) per
geometry.setAttribute( 'position', new THREE.BufferAttribute( positions, 3 ) );

// draw range
const drawCount = 2; // draw the first 2 points, only
geometry.setDrawRange( 0, drawCount );

// material
const material = new THREE.LineBasicMaterial( { color: 0xff0000 } );

// line
const line = new THREE.Line( geometry, material );
scene.add( line );
```

javascript

```javascript
const positionAttribute = line.geometry.getAttribute( 'position' );

let x = 0, y = 0, z = 0;

for ( let i = 0; i < positionAttribute.count; i ++ ) {

    positionAttribute.setXYZ( i, x, y, z );

    x += ( Math.random() - 0.5 ) * 30;
    y += ( Math.random() - 0.5 ) * 30;
    z += ( Math.random() - 0.5 ) * 30;

}
```

javascript

```javascript
line.geometry.setDrawRange( 0, newValue );
```

javascript

```javascript
positionAttribute.needsUpdate = true; // required after the first render
```

javascript

```
line.geometry.computeBoundingBox();
line.geometry.computeBoundingSphere();
```

Examples

WebGL / custom / attributes

WebGL / buffergeometry / custom / attributes / particles

Materials

All uniforms values can be changed freely (e.g. colors, textures, opacity, etc), values are sent to the shader every frame.

Also GLstate related parameters can change any time (depthTest, blending, polygonOffset, etc).

The following properties can't be easily changed at runtime (once the material is rendered at least once):

- numbers and types of uniforms
- presence or not of
- texture
- fog
- vertex colors
- morphing
- shadow map
- alpha test
- transparent

Changes in these require building of new shader program. You'll need to set

`material.needsUpdate = true`

Bear in mind this might be quite slow and induce jerkiness in framerate (especially on Windows, as shader compilation is slower in DirectX than OpenGL).

For smoother experience you can emulate changes in these features to some degree by having "dummy" values like zero intensity lights, white textures, or zero density fog.

You can freely change the material used for geometry chunks, however you cannot change how an object is divided into chunks (according to face materials).

If you need to have different configurations of materials during runtime:

If the number of materials / chunks is small, you could pre-divide the object beforehand (e.g. hair / face / body / upper clothes / trousers for a human, front / sides / top / glass / tire / interior for a car).

If the number is large (e.g. each face could be potentially different), consider a different solution, such as using attributes / textures to drive different per-face look.

Examples

WebGL / materials / car

WebGL / webgl_postprocessing / dof

Textures

Image, canvas, video and data textures need to have the following flag set if they are changed:

`texture.needsUpdate = true;`

Render targets update automatically.

```javascript
texture.needsUpdate = true;
```

Examples

WebGL / materials / video

WebGL / rtt

Cameras

A camera's position and target is updated automatically. If you need to change

- fov
- aspect
- near
- far

then you'll need to recompute the projection matrix:

```javascript
camera.aspect = window.innerWidth / window.innerHeight;
camera.updateProjectionMatrix();
```

InstancedMesh

`InstancedMesh` is a class for conveniently access instanced rendering in `three.js`. Certain library features like view frustum culling or ray casting rely on up-to-date bounding volumes (bounding sphere and bounding box). Because of the way how `InstancedMesh` works, the class has its own `boundingBox` and `boundingSphere` properties that supersede the bounding volumes on geometry level.

Similar to geometries you have to recompute the bounding box and sphere whenever you change the underlying data. In context of `InstancedMesh`, that happens when you transform instances via `setMatrixAt()`. You can use the same pattern like with geometries.

```javascript
instancedMesh.computeBoundingBox();
instancedMesh.computeBoundingSphere();
```

SkinnedMesh

`SkinnedMesh` follows the same principles like `InstancedMesh` in context of bounding volumes. Meaning the class has its own version of `boundingBox` and `boundingSphere` to correctly enclose animated meshes. When calling `computeBoundingBox()` and `computeBoundingSphere()`, the class computes the respective bounding volumes

based on the current bone transformation (or in other words the current animation state).

Cleanup

GUIDE

Source: https://threejs.org/manual/en/cleanup.html

A guide to managing memory and cleaning up Three.js resources (geometries, textures, materials, loaded files) by implementing a ResourceTracker class to automate disposal.

Cleanup

Three.js apps often use lots of memory. A 3D model might be 1 to 20 meg memory for all of its vertices. A model might use many textures that even if they are compressed into jpg files they have to be expanded to their uncompressed form to use. Each 1024x1024 texture takes 4 to 6meg of memory.

Most three.js apps load resources at init time and then use those resources forever until the page is closed. But, what if you want to load and change resources over time?

Unlike most JavaScript, three.js can not automatically clean these resources up. The browser will clean them up if you switch pages but otherwise it's up to you to manage them. This is an issue of how WebGL is designed and so three.js has no recourse but to pass on the responsibility to free resources back to you.

You free three.js resource this by calling the `dispose` function on textures, geometries, and materials.

You could do this manually. At the start you might create some of these resources

```
const boxGeometry = new THREE.BoxGeometry(...);
const boxTexture = textureLoader.load(...);
const boxMaterial = new THREE.MeshPhongMaterial({map: texture});
```

and then when you're done with them you'd free them

```
boxGeometry.dispose();
boxTexture.dispose();
boxMaterial.dispose();
```

As you use more and more resources that would get more and more tedious.

To help remove some of the tedium let's make a class to track the resources. We'll then ask that class to do the cleanup for us.

Here's a first pass at such a class

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (resource.dispose) {
      this.resources.add(resource);
    }
    return resource;
  }
  untrack(resource) {
    this.resources.delete(resource);
  }
  dispose() {
    for (const resource of this.resources) {
      resource.dispose();
    }
    this.resources.clear();
  }
}
```

Let's use this class with the first example from the article on textures. We can create an instance of this class

```
const resTracker = new ResourceTracker();
```

and then just to make it easier to use let's create a bound function for the `track` method

```
const resTracker = new ResourceTracker();
+const track = resTracker.track.bind(resTracker);
```

Now to use it we just need to call `track` with for each geometry, texture, and material we create

```
const boxWidth = 1;
const boxHeight = 1;
const boxDepth = 1;
-const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
+const geometry = track(new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth));

const cubes = [];  // an array we can use to rotate the cubes
const loader = new THREE.TextureLoader();

-const material = new THREE.MeshBasicMaterial({
-  map: loader.load('resources/images/wall.jpg'),
-});
+const material = track(new THREE.MeshBasicMaterial({
+  map: track(loader.load('resources/images/wall.jpg')),
+}));
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
cubes.push(cube);  // add to our list of cubes to rotate
```

And then to free them we'd want to remove the cubes from the scene and then call `resTracker.dispose`

```
for (const cube of cubes) {
  scene.remove(cube);
}
cubes.length = 0;  // clears the cubes array
resTracker.dispose();
```

That would work but I find having to remove the cubes from the scene kind of tedious. Let's add that functionality to the `ResourceTracker`.

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
-    if (resource.dispose) {
+    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    return resource;
  }
  untrack(resource) {
    this.resources.delete(resource);
  }
  dispose() {
    for (const resource of this.resources) {
-      resource.dispose();
+      if (resource instanceof THREE.Object3D) {
+        if (resource.parent) {
+          resource.parent.remove(resource);
+        }
+      }
+      if (resource.dispose) {
+        resource.dispose();
+      }
+    }
    this.resources.clear();
  }
}
```

And now we can track the cubes

```
const material = track(new THREE.MeshBasicMaterial({
  map: track(loader.load('resources/images/wall.jpg')),
}));
const cube = track(new THREE.Mesh(geometry, material));
scene.add(cube);
cubes.push(cube);  // add to our list of cubes to rotate
```

We no longer need the code to remove the cubes from the scene.

```
-for (const cube of cubes) {
-  scene.remove(cube);
-}
cubes.length = 0;  // clears the cube array
resTracker.dispose();
```

Let's arrange this code so that we can re-add the cube, texture, and material.

```
const scene = new THREE.Scene();
*const cubes = [];  // just an array we can use to rotate the cubes

+function addStuffToScene() {
  const resTracker = new ResourceTracker();
  const track = resTracker.track.bind(resTracker);

  const boxWidth = 1;
  const boxHeight = 1;
  const boxDepth = 1;
  const geometry = track(new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth));

  const loader = new THREE.TextureLoader();

  const material = track(new THREE.MeshBasicMaterial({
    map: track(loader.load('resources/images/wall.jpg')),
  }));
  const cube = track(new THREE.Mesh(geometry, material));
  scene.add(cube);
  cubes.push(cube);  // add to our list of cubes to rotate
+  return resTracker;
+}
```

And then let's write some code to add and remove things over time.

```
function waitSeconds(seconds = 0) {
  return new Promise(resolve => setTimeout(resolve, seconds * 1000));
}

async function process() {
  for (;;) {
    const resTracker = addStuffToScene();
    await wait(2);
    cubes.length = 0;  // remove the cubes
    resTracker.dispose();
    await wait(1);
  }
}
process();
```

This code will create the cube, texture and material, wait for 2 seconds, then dispose of them and wait for 1 second and repeat.

click here to open in a separate window

So that seems to work.

For a loaded file though it's a little more work. Most loaders only return an `Object3D` as a root of the hierarchy of objects they load so we need to discover what all the resources are.

Let's update our `ResourceTracker` to try to do that.

First we'll check if the object is an `Object3D` then track its geometry, material, and children

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
+    if (resource instanceof THREE.Object3D) {
+      this.track(resource.geometry);
+      this.track(resource.material);
+      this.track(resource.children);
+    }
    return resource;
  }
  ...
}
```

Now, because any of `resource.geometry`, `resource.material`, and `resource.children` might be null or undefined we'll check at the top of `track`.

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
+    if (!resource) {
+      return resource;
+    }

    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    if (resource instanceof THREE.Object3D) {
      this.track(resource.geometry);
      this.track(resource.material);
      this.track(resource.children);
    }
    return resource;
  }
  ...
}
```

Also because `resource.children` is an array and because `resource.material` can be an array let's check for arrays

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (!resource) {
      return resource;
    }

+    // handle children and when material is an array of materials.
+    if (Array.isArray(resource)) {
+      resource.forEach(resource => this.track(resource));
+      return resource;
+    }

    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    if (resource instanceof THREE.Object3D) {
      this.track(resource.geometry);
      this.track(resource.material);
      this.track(resource.children);
    }
    return resource;
  }
  ...
}
```

And finally we need to walk the properties and uniforms of a material looking for textures.

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (!resource) {
      return resource;
    }

*    // handle children and when material is an array of materials or
*    // uniform is array of textures
    if (Array.isArray(resource)) {
      resource.forEach(resource => this.track(resource));
      return resource;
    }

    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    if (resource instanceof THREE.Object3D) {
      this.track(resource.geometry);
      this.track(resource.material);
      this.track(resource.children);
-    }
+    } else if (resource instanceof THREE.Material) {
+      // We have to check if there are any textures on the material
+      for (const value of Object.values(resource)) {
+        if (value instanceof THREE.Texture) {
+          this.track(value);
+        }
+      }
+      // We also have to check if any uniforms reference textures or arrays of tex
+      if (resource.uniforms) {
+        for (const value of Object.values(resource.uniforms)) {
+          if (value) {
+            const uniformValue = value.value;
+            if (uniformValue instanceof THREE.Texture ||
+                Array.isArray(uniformValue)) {
+              this.track(uniformValue);
+            }
+          }
+        }
+      }
+    }
    return resource;
  }
  ...
}
```

And with that let's take an example from the article on loading gltf files and make it load and free files.

```
const gltfLoader = new GLTFLoader();
function loadGLTF(url) {
  return new Promise((resolve, reject) => {
    gltfLoader.load(url, resolve, undefined, reject);
  });
}

function waitSeconds(seconds = 0) {
  return new Promise(resolve => setTimeout(resolve, seconds * 1000));
}

const fileURLs = [
  'resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf',
  'resources/models/3dbustchallange_submission/scene.gltf',
  'resources/models/mountain_landscape/scene.gltf',
  'resources/models/simple_house_scene/scene.gltf',
];

async function loadFiles() {
  for (;;) {
    for (const url of fileURLs) {
      const resMgr = new ResourceTracker();
      const track = resMgr.track.bind(resMgr);
      const gltf = await loadGLTF(url);
      const root = track(gltf.scene);
      scene.add(root);

      // compute the box that contains all the stuff
      // from root and below
      const box = new THREE.Box3().setFromObject(root);

      const boxSize = box.getSize(new THREE.Vector3()).length();
      const boxCenter = box.getCenter(new THREE.Vector3());

      // set the camera to frame the box
      frameArea(boxSize * 1.1, boxSize, boxCenter, camera);

      await waitSeconds(2);
      renderer.render(scene, camera);

      resMgr.dispose();

      await waitSeconds(1);

    }
  }
}
loadFiles();
```

and we get

click here to open in a separate window

Some notes about the code.

If we wanted to load 2 or more files at once and free them at anytime we would use one `ResourceTracker` per file.

Above we are only tracking `gltf.scene` right after loading. Based on our current implementation of `ResourceTracker` that will track all the resources just loaded. If we added more things to the scene we need to decide whether or not to track them.

For example let's say after we loaded a character we put a tool in their hand by making the tool a child of their hand. As it is that tool will not be freed. I'm guessing more often than not this is what we want.

That brings up a point. Originally when I first wrote the `ResourceTracker` above I walked through everything inside the `dispose` method instead of `track`. It was only later as I thought about the tool as a child of hand case above that it became clear that tracking exactly what to free in `track` was more flexible and arguably more correct since we could then track what was loaded from the file rather than just freeing the state of the scene graph later.

I honestly am not 100% happy with `ResourceTracker`. Doing things this way is not common in 3D engines. We shouldn't have to guess what resources were loaded, we should know. It would be nice if three.js changed so that all file loaders returned some standard object with references to all the resources loaded. At least at the moment, three.js doesn't give us any more info when loading a scene so this solution seems to work.

I hope you find this example useful or at least a good reference for what is required to free resources in three.js

javascript

```
const boxGeometry = new THREE.BoxGeometry(...);
const boxTexture = textureLoader.load(...);
const boxMaterial = new THREE.MeshPhongMaterial({map: texture});
```

javascript

```
boxGeometry.dispose();
boxTexture.dispose();
boxMaterial.dispose();
```

javascript

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (resource.dispose) {
      this.resources.add(resource);
    }
    return resource;
  }
  untrack(resource) {
    this.resources.delete(resource);
  }
  dispose() {
    for (const resource of this.resources) {
      resource.dispose();
    }
    this.resources.clear();
  }
}
```

javascript

```
const resTracker = new ResourceTracker();
```

javascript

```
const resTracker = new ResourceTracker();
+const track = resTracker.track.bind(resTracker);
```

javascript

```
const boxWidth = 1;
const boxHeight = 1;
const boxDepth = 1;
-const geometry = new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth);
+const geometry = track(new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth));

const cubes = [];  // an array we can use to rotate the cubes
const loader = new THREE.TextureLoader();

-const material = new THREE.MeshBasicMaterial({
-  map: loader.load('resources/images/wall.jpg'),
-});
+const material = track(new THREE.MeshBasicMaterial({
+  map: track(loader.load('resources/images/wall.jpg')),
+}));
const cube = new THREE.Mesh(geometry, material);
scene.add(cube);
cubes.push(cube);  // add to our list of cubes to rotate
```

javascript

```
for (const cube of cubes) {
  scene.remove(cube);
}
cubes.length = 0;  // clears the cubes array
resTracker.dispose();
```

javascript

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
-    if (resource.dispose) {
+    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    return resource;
  }
  untrack(resource) {
    this.resources.delete(resource);
  }
  dispose() {
    for (const resource of this.resources) {
-      resource.dispose();
+      if (resource instanceof THREE.Object3D) {
+        if (resource.parent) {
+          resource.parent.remove(resource);
+        }
+      }
+      if (resource.dispose) {
+        resource.dispose();
+      }
+    }
    this.resources.clear();
  }
}
```

javascript

```
const material = track(new THREE.MeshBasicMaterial({
  map: track(loader.load('resources/images/wall.jpg')),
}));
const cube = track(new THREE.Mesh(geometry, material));
scene.add(cube);
cubes.push(cube);  // add to our list of cubes to rotate
```

javascript

```
-for (const cube of cubes) {
-  scene.remove(cube);
-}
cubes.length = 0;  // clears the cube array
resTracker.dispose();
```

javascript

```
const scene = new THREE.Scene();
*const cubes = [];  // just an array we can use to rotate the cubes

+function addStuffToScene() {
  const resTracker = new ResourceTracker();
  const track = resTracker.track.bind(resTracker);

  const boxWidth = 1;
  const boxHeight = 1;
  const boxDepth = 1;
  const geometry = track(new THREE.BoxGeometry(boxWidth, boxHeight, boxDepth));

  const loader = new THREE.TextureLoader();

  const material = track(new THREE.MeshBasicMaterial({
    map: track(loader.load('resources/images/wall.jpg')),
  }));
  const cube = track(new THREE.Mesh(geometry, material));
  scene.add(cube);
  cubes.push(cube);  // add to our list of cubes to rotate
+  return resTracker;
+}
```

javascript

```
function waitSeconds(seconds = 0) {
  return new Promise(resolve => setTimeout(resolve, seconds * 1000));
}

async function process() {
  for (;;) {
    const resTracker = addStuffToScene();
    await wait(2);
    cubes.length = 0;  // remove the cubes
    resTracker.dispose();
    await wait(1);
  }
}
process();
```

javascript

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
+    if (resource instanceof THREE.Object3D) {
+      this.track(resource.geometry);
+      this.track(resource.material);
+      this.track(resource.children);
+    }
    return resource;
  }
  ...
}
```

javascript

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
+    if (!resource) {
+      return resource;
+    }

    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    if (resource instanceof THREE.Object3D) {
      this.track(resource.geometry);
      this.track(resource.material);
      this.track(resource.children);
    }
    return resource;
  }
  ...
}
```

javascript

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (!resource) {
      return resource;
    }

+    // handle children and when material is an array of materials.
+    if (Array.isArray(resource)) {
+      resource.forEach(resource => this.track(resource));
+      return resource;
+    }

    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    if (resource instanceof THREE.Object3D) {
      this.track(resource.geometry);
      this.track(resource.material);
      this.track(resource.children);
    }
    return resource;
  }
  ...
}
```

javascript

```
class ResourceTracker {
  constructor() {
    this.resources = new Set();
  }
  track(resource) {
    if (!resource) {
      return resource;
    }

*    // handle children and when material is an array of materials or
*    // uniform is array of textures
    if (Array.isArray(resource)) {
      resource.forEach(resource => this.track(resource));
      return resource;
    }

    if (resource.dispose || resource instanceof THREE.Object3D) {
      this.resources.add(resource);
    }
    if (resource instanceof THREE.Object3D) {
      this.track(resource.geometry);
      this.track(resource.material);
      this.track(resource.children);
-    }
+    } else if (resource instanceof THREE.Material) {
+      // We have to check if there are any textures on the material
+      for (const value of Object.values(resource)) {
+        if (value instanceof THREE.Texture) {
+          this.track(value);
+        }
+      }
+      // We also have to check if any uniforms reference textures or arrays of tex
+      if (resource.uniforms) {
+        for (const value of Object.values(resource.uniforms)) {
+          if (value) {
+            const uniformValue = value.value;
+            if (uniformValue instanceof THREE.Texture ||
+                Array.isArray(uniformValue)) {
+              this.track(uniformValue);
+            }
+          }
+        }
+      }
+    }
    return resource;
  }
  ...
}
```

javascript

```
const gltfLoader = new GLTFLoader();
function loadGLTF(url) {
  return new Promise((resolve, reject) => {
    gltfLoader.load(url, resolve, undefined, reject);
  });
}

function waitSeconds(seconds = 0) {
  return new Promise(resolve => setTimeout(resolve, seconds * 1000));
}

const fileURLs = [
  'resources/models/cartoon_lowpoly_small_city_free_pack/scene.gltf',
  'resources/models/3dbustchallange_submission/scene.gltf',
  'resources/models/mountain_landscape/scene.gltf',
  'resources/models/simple_house_scene/scene.gltf',
];

async function loadFiles() {
  for (;;) {
    for (const url of fileURLs) {
      const resMgr = new ResourceTracker();
      const track = resMgr.track.bind(resMgr);
      const gltf = await loadGLTF(url);
      const root = track(gltf.scene);
      scene.add(root);

      // compute the box that contains all the stuff
      // from root and below
      const box = new THREE.Box3().setFromObject(root);

      const boxSize = box.getSize(new THREE.Vector3()).length();
      const boxCenter = box.getCenter(new THREE.Vector3());

      // set the camera to frame the box
      frameArea(boxSize * 1.1, boxSize, boxCenter, camera);

      await waitSeconds(2);
      renderer.render(scene, camera);

      resMgr.dispose();

      await waitSeconds(1);

    }
  }
}
loadFiles();
```

Debugging - GLSL

GUIDE

Source: https://threejs.org/manual/en/debugging-glsl.html

A guide on debugging GLSL shaders in Three.js, offering practical tips for troubleshooting fragment and vertex shader issues through visualization techniques,

simplified shaders, and developer tools.

Debugging - GLSL

This site so far does not teach GLSL just like it does not teach JavaScript. Those are really large topics. If you want to learn GLSL consider checking out these articles as a starting place.

If you already know GLSL then here are a few tips for debugging.

When I'm making a new GLSL shader and nothing appears generally the first thing I do is change the fragment shader to return a solid color. For example at the very bottom of the shader I might put

```
void main() {

  ...

  gl_FragColor = vec4(1, 0, 0, 1);  // red
}
```

If I see the object I was trying to draw then I know the issue is related to my fragment shader. It could be anything like bad textures, uninitialized uniforms, uniforms with the wrong values but at least I have a direction to look.

To test some of those I might start trying to draw some of the inputs. For example if I'm using normals in the fragment shader then I might add

```
gl_FragColor = vec4(vNormal * 0.5 + 0.5, 1);
```

Normals go from -1 to +1 so by multiplying by 0.5 and adding 0.5 we get values that go from 0.0 to 1.0 which makes them useful for colors.

Try this with some things you know work and you'll start getting an idea of what normals *normally* look like. If your normals don't look normal then you have some clue where to look. If you're manipulating normals in the fragments shader you can use the same technique to draw the result of that manipulation.

Similarly if we're using textures there will be texture coordinates and we can draw them with something like

```
gl_FragColor = vec4(fract(vUv), 0, 1);
```

The `fract` is there in case we're using texture coordinates that go outside the 0 to 1 range. This is common if `texture.repeat` is set to something greater than 1.

You can do similar things for all values in your fragment shader. Figure out what their range is likely to be, add some code to set `gl_FragColor` with that range scaled to 0.0 to 1.0

To check textures try a `CanvasTexture` or a `DataTexture` that you know works.

Conversely, if after setting `gl_FragColor` to red I still see nothing then I have a hint my issue might be in the direction of the things related to the vertex shader. Some matrices might be wrong or my attributes might have bad data or be setup incorrectly.

I'd first look at the matrices. I might put a breakpoint right after my call to `renderer.render(scene, camera)` and then start expanding things in the inspector. Is the camera's world matrix and projection matrix at least not full of `NaN`s? Expanding the scene and looking at its `children` I'd check that the world matrices look reasonable (no `NaN`s) and last 4 values of each matrix look reasonable for my scene. If I expect my scene to be 50x50x50 units and some matrix shows 552352623.123 clearly something is wrong there.

Just like we did for the fragment shader we can also draw values from the vertex shader by passing them to the fragment shader. Declare a varying in both and pass the value you're not sure is correct. In fact if my shader use using normals I'll change the fragment shader to display them like is mentioned above and then just set `vNormal` to the value I want to display but scaled so the values go from 0.0 to 1.0. I then look at the results and see if they fit my expectations.

Another good thing to do is use a simpler shader. Can you draw your data with `MeshBasicMaterial`? If you can then try it and make sure it shows up as expected.

If not what's the simplest vertex shader that will let you visualize your geometry? Usually it's as simple as

```
gl_Position = projection * modelView * vec4(position.xyz, 1);
```

If that works start adding in your changes a little at a time.

Yet another thing you can do is use the Shader Editor extension for Chrome or similar for other browsers. It's a great way to look at how other shaders are working. It's also good as you can make some of the changes suggested above live while the code is running.

```glsl
void main() {

  ...

  gl_FragColor = vec4(1, 0, 0, 1);  // red
}
```

```glsl
gl_FragColor = vec4(vNormal * 0.5 + 0.5, 1);
```

```glsl
gl_FragColor = vec4(fract(vUv), 0, 1);
```

```glsl
gl_Position = projection * modelView * vec4(position.xyz, 1);
```

Compatibility & FAQ

WebGL Compatibility Check

GUIDE

Source: https://threejs.org/manual/en/webgl-compatibility-check.html

Explains how to check for WebGL 2 support in Three.js and display a fallback message to users when it is unavailable.

WebGL Compatibility Check

Even though this is becoming less and less of a problem, some devices or browsers may still not support WebGL 2. The following method allows you to check if it is supported and display a message to the user if it is not. Import the WebGL support detection module, and run the following before attempting to render anything.

javascript

```javascript
import WebGL from 'three/addons/capabilities/WebGL.js';

if ( WebGL.isWebGL2Available() ) {

  // Initiate function or other initializations here
  animate();

} else {

  const warning = WebGL.getWebGL2ErrorMessage();
  document.getElementById( 'container' ).appendChild( warning );

}
```

FAQ

GUIDE

Source: https://threejs.org/manual/en/faq.html

A frequently asked questions page covering best-supported 3D model formats, viewport tags, scene scaling on resize, face culling, input validation, and Node.js usage in three.js.

Which 3D model format is best supported?

The recommended format for importing and exporting assets is glTF (GL Transmission Format). Because glTF is focused on runtime asset delivery, it is compact to transmit and fast to load.

three.js provides loaders for many other popular formats like FBX, Collada or OBJ as well. Nevertheless, you should always try to establish a glTF based workflow in your projects first.

Why are there meta viewport tags in examples?

These tags control viewport size and scale for mobile browsers (where page content may be rendered at different size than visible viewport).

```html
<meta name="viewport" content="width=device-width, user-scalable=no, minimum-scale=
```

How can scene scale be preserved on resize?

We want all objects, regardless of their distance from the camera, to appear the same size, even as the window is resized.

The key equation to solving this is this formula for the visible height at a given distance:

If we increase the window height by a certain percentage, then what we want is the visible height at all distances to increase by the same percentage.

This can not be done by changing the camera position. Instead you have to change the camera field-of-view.

```javascript
visible_height = 2 * Math.tan( ( Math.PI / 180 ) * camera.fov / 2 ) * distance_from
```

Why is part of my object invisible?

This could be because of face culling. Faces have an orientation that decides which side is which. And the culling removes the backside in normal circumstances.

To see if this is your problem, change the material side to THREE.DoubleSide.

```javascript
material.side = THREE.DoubleSide
```

Why does three.js sometimes return strange results for invalid inputs?

For performance reasons, three.js doesn't validate inputs in most cases. It's your app's responsibility to make sure that all inputs are valid.

Can I use three.js in Node.js?

Because three.js is built for the web, it depends on browser and DOM APIs that don't always exist in Node.js. Some of these issues can be avoided by using shims like

headless-gl and jsdom-global, or by replacing components like `TextureLoader` with custom alternatives. Other DOM APIs may be deeply intertwined with the code that uses them, and will be harder to work around. We welcome simple and maintainable pull requests to improve Node.js support, but recommend opening an issue to discuss your improvements first.

Reference

Libraries and Plugins

REFERENCE

Source: https://threejs.org/manual/en/libraries-and-plugins.html

A community-maintained list of externally developed compatible libraries and plugins for three.js, organized by category including physics, postprocessing, file formats, geometry, 3D text, particle systems, IK, game AI, and wrappers/frameworks.

Libraries and Plugins

Listed here are externally developed compatible libraries and plugins for three.js. This list and the associated packages are maintained by the community and not guaranteed to be up to date. If you'd like to update this list make a PR!

Physics

Postprocessing

In addition to the official three.js postprocessing effects, support for some additional effects and frameworks are available through external libraries.

Intersection and Raycast Performance

Path Tracing

File Formats

In addition to the official three.js loaders, support for some additional formats is available through external libraries.

Geometry

3D Text and Layout

Particle Systems

Inverse Kinematics

Game AI

Wrappers and Frameworks

Uniform Types

REFERENCE

Source: https://threejs.org/manual/en/uniform-types.html

Documents the mapping between GLSL uniform types and their corresponding JavaScript types in three.js, including support for uniform structures and arrays.

Uniform Types

Each uniform must have a `value` property. The type of the value must correspond to the type of the uniform variable in the GLSL code as specified for the primitive GLSL types in the table below. Uniform structures and arrays are also supported. GLSL arrays of primitive type must either be specified as an array of the corresponding THREE objects or as a flat array containing the data of all the objects. In other words; GLSL primitives in arrays must not be represented by arrays. This rule does not apply transitively. An array of `vec2` arrays, each with a length of five vectors, must be an array of arrays, of either five `Vector2` objects or ten `number`s.

GLSL type	JavaScript type
int	Number
uint	Number
float	Number
bool	Boolean
bool	Number
vec2	Vector2
vec2	Float32Array (*)
vec2	Array (*)
vec3	Vector3
vec3	Color
vec3	Float32Array (*)
vec3	Array (*)
vec4	Vector4
vec4	Quaternion
vec4	Float32Array (*)
vec4	Array (*)
mat2	Float32Array (*)
mat2	Array (*)

GLSL type	JavaScript type
mat3	Matrix3
mat3	Float32Array (*)
mat3	Array (*)
mat4	Matrix4
mat4	Float32Array (*)
mat4	Array (*)
ivec2, bvec2	Float32Array (*)
ivec2, bvec2	Array (*)
ivec3, bvec3	Int32Array (*)
ivec3, bvec3	Array (*)
ivec4, bvec4	Int32Array (*)
ivec4, bvec4	Array (*)
sampler2D	Texture
samplerCube	CubeTexture

(*) Same for an (innermost) array (dimension) of the same GLSL type, containing the components of all vectors or matrices in the array.

Structured Uniforms

Sometimes you want to organize uniforms as `structs` in your shader code. The following style must be used so `three.js` is able to process structured uniform data.

This definition can be mapped on the following GLSL code:

```javascript
uniforms = {
  data: {
    value: {
      position: new Vector3(),
      direction: new Vector3( 0, 0, 1 )
    }
  }
};
```

```glsl
struct Data {
  vec3 position;
  vec3 direction;
};
uniform Data data;
```

Structured Uniforms with Arrays

It's also possible to manage `structs` in arrays. The syntax for this use case looks like so:

This definition can be mapped on the following GLSL code:

```javascript
const entry1 = {
  position: new Vector3(),
  direction: new Vector3( 0, 0, 1 )
};
const entry2 = {
  position: new Vector3( 1, 1, 1 ),
  direction: new Vector3( 0, 1, 0 )
};

uniforms = {
  data: {
    value: [ entry1, entry2 ]
  }
};
```

```glsl
struct Data {
  vec3 position;
  vec3 direction;
};
uniform Data data[ 2 ];
```

Useful Links

REFERENCE

Source: https://threejs.org/manual/en/useful-links.html

A curated collection of links to tutorials, courses, examples, tools, forums, and other resources useful when learning three.js.

Introduction

The following is a collection of links that you might find useful when learning three.js.

If you find something that you'd like to add here, or think that one of the links below is no longer relevant or working, feel free to click the 'edit' button in the bottom right and make some changes!

Note also that as three.js is under rapid development, a lot of these links will contain information that is out of date - if something isn't working as you'd expect or as one of these links says it should, check the browser console for warnings or errors. Also check the relevant docs pages.

Help forums

Three.js officially uses the forum (https://discourse.threejs.org/) and Stack Overflow (http://stackoverflow.com/tags/three.js/info) for help requests. If you need assistance with something, that's the place to go. Do NOT open an issue on Github for help requests.

Tutorials and courses

Getting started with three.js

- Three.js Fundamentals starting lesson (https://threejs.org/manual/#en/fundamentals)
- Beginning with 3D WebGL (https://codepen.io/rachsmith/post/beginning-with-3d-webgl-pt-1-the-scene) by Rachel Smith (https://codepen.io/rachsmith/).
- Animating scenes with WebGL and three.js (https://www.august.com.au/blog/animating-scenes-with-webgl-three-js/)

More extensive / advanced articles and courses

- Three Journey (https://threejs-journey.com/) Course by Bruno Simon (https://bruno-simon.com/) - Teaches beginners how to use Three.js step by step
- Discover three.js (https://discoverthreejs.com/)
- Collection of tutorials (http://blog.cjgammon.com/) by CJ Gammon (http://www.cjgammon.com/).
- Glossy spheres in three.js (https://medium.com/soffritti.pierfrancesco/glossy-spheres-in-three-js-bfd2785d4857).
- Interactive 3D Graphics (https://www.udacity.com/course/interactive-3d-graphics--cs291) - a free course on Udacity that teaches the fundamentals of 3D Graphics, and uses three.js as its coding tool.
- Aerotwist (https://aerotwist.com/tutorials/) tutorials by Paul Lewis (https://github.com/paullewis/).
- Three.js Bookshelf (https://discourse.threejs.org/t/three-js-bookshelf/2468) - Looking for more resources about three.js or computer graphics in general? Check out the selection of literature recommended by the community.

News and Updates

- Three.js on Twitter (https://twitter.com/hashtag/threejs)
- Three.js on reddit (http://www.reddit.com/r/threejs/)
- WebGL on reddit (http://www.reddit.com/r/webgl/)

Examples

- three-seed (https://github.com/edwinwebb/three-seed/) - three.js starter project with ES6 and Webpack
- Professor Stemkoskis Examples (http://stemkoski.github.io/Three.js/index.html) - a collection of beginner friendly examples built using three.js r60.
- Official three.js examples (https://threejs.org/examples/) - these examples are maintained as part of the three.js repository, and always use the latest version of three.js.

- Official three.js dev branch examples (https://raw.githack.com/mrdoob/three.js/dev/examples/) - Same as the above, except these use the dev branch of three.js, and are used to check that everything is working as three.js being is developed.

Tools

- physgl.org (https://github.com/tbensky/physgl) - JavaScript front-end with wrappers to three.js, to bring WebGL graphics to students learning physics and math.
- Whitestorm.js (https://whsjs.readme.io/) – Modular three.js framework with AmmoNext physics plugin.
- Three.js Inspector (http://zz85.github.io/zz85-bookmarklets/threelabs.html)
- ThreeNodes.js (http://idflood.github.io/ThreeNodes.js/).
- vscode shader (https://marketplace.visualstudio.com/items?itemName=slevesque.shader) - Syntax highlighter for shader language.
- vscode comment-tagged-templates (https://marketplace.visualstudio.com/items?itemName=bierner.comment-tagged-templates) - Syntax highlighting for tagged template strings using comments to shader language, like: glsl.js.
- WebXR-emulator-extension (https://github.com/MozillaReality/WebXR-emulator-extension)

WebGL References

- webgl-reference-card.pdf (https://www.khronos.org/files/webgl/webgl-reference-card-1_0.pdf) - Reference of all WebGL and GLSL keywords, terminology, syntax and definitions.

Old Links

These links are kept for historical purposes - you may still find them useful, but be warned that they may have information relating to very old versions of three.js.

- AlterQualia at WebGL Camp 3 (https://www.youtube.com/watch?v=Dir4KO9RdhM)
- Yomotsus Examples (http://yomotsu.github.io/threejs-examples/) - a collection of examples using three.js r45.

- Introduction to Three.js (http://fhtr.org/BasicsOfThreeJS/#1) by Ilmari Heikkinen (http://github.com/kig/) (slideshow).
- WebGL and Three.js (http://www.slideshare.net/yomotsu/webgl-and-threejs) by Akihiro Oyamada (http://github.com/yomotsu) (slideshow).
- Trigger Rally (https://www.youtube.com/watch?v=VdQnOaolrPA) by jareiko (https://github.com/jareiko) (video).
- ThreeFab (http://blackjk3.github.io/threefab/) - scene editor, maintained up until around three.js r50.
- Max to Three.js workflow tips and tricks (http://bkcore.com/blog/3d/webgl-three-js-workflow-tips.html) by BKcore (https://github.com/BKcore)
- A whirlwind look at Three.js (http://12devsofxmas.co.uk/2012/01/webgl-and-three-js/) by Paul King (http://github.com/nrocy)
- Animated selective glow in Three.js (http://bkcore.com/blog/3d/webgl-three-js-animated-selective-glow.html) by BKcore (https://github.com/BKcore)
- Building A Physics Simulation Environment (http://www.natural-science.or.jp/article/20120220155529.php) - three.js tutorial in Japanese

Material Feature Table

REFERENCE

Source: https://threejs.org/manual/en/material-table.html

A reference table showing which properties and features are supported by each of the five main Mesh materials in three.js (Basic, Lambert, Phong, Standard, Physical).

Material Feature Table

The most common materials in three.js are the Mesh materials. Here is a table showing which material support which features.

The table below compares feature support across the five main Mesh materials. Each column header links to the corresponding material documentation; each table cell with a bullet (•) links to that property's documentation page.

Materials compared: Basic, Lambert, Phong, Standard, Physical.

Feature comparison (• indicates support):

Feature	Basic	Lambert	Phong	Standard	Physical
alphaMap	•	•	•	•	•
anisotropy					•
anisotropyMap					•
anisotropyRotation					•
aoMap	•	•	•	•	•
aoMapIntensity	•	•	•	•	•
attenuationColor					•
attenuationDistance					•
bumpMap		•	•	•	•
bumpScale		•	•	•	•
clearcoat					•
clearcoatMap					•
clearcoatNormalMap					•
clearcoatNormalScale					•
clearcoatRoughness					•
clearcoatRoughnessMap					•
color	•	•	•	•	•
combine	•	•	•		

Feature	Basic	Lambert	Phong	Standard	Physical
displacementBias		•	•	•	•
displacementMap		•	•	•	•
displacementScale		•	•	•	•
emissive		•	•	•	•
emissiveIntensity		•	•	•	•
emissiveMap		•	•	•	•
envMap	•	•	•	•	•
envMapIntensity				•	•
envMapRotation	•	•	•	•	•
flatShading		•	•	•	•
fog	•	•	•	•	•
ior					•
iridescence					•
iridescenceIOR					•
iridescenceMap					•
iridescenceThicknessMap					•
iridescenceThicknessRange					•
lightMap	•	•	•	•	•

Feature	Basic	Lambert	Phong	Standard	Physical
lightMapIntensity	•	•	•	•	•
map	•	•	•	•	•
metalness				•	•
metalnessMap				•	•
normalMap		•	•	•	•
normalMapType		•	•	•	•
normalScale		•	•	•	•
reflectivity	•	•	•		•
refractionRatio	•	•	•		
roughness				•	•
roughnessMap				•	•
sheen					•
sheenColor					•
sheenColorMap					•
sheenRoughness					•
sheenRoughnessMap					•
shininess			•		
specular			•		

Feature	Basic	Lambert	Phong	Standard	Physical
specularColor					•
specularColorMap					•
specularIntensity					•
specularIntensityMap					•
specularMap	•	•	•		
thickness					•
thicknessMap					•
transmission					•
transmissionMap					•
wireframe	•	•	•	•	•
wireframeLinecap	•	•	•	•	•
wireframeLinejoin	•	•	•	•	•
wireframeLinewidth	•	•	•	•	•