Notes on THREE.js webGL interface

So now you want to learn how to create cool 3D objects in a web browser? Well, you can always use webGL directly. But that’s a nightmare and would take you hundreds of lines for one piece of cube. You aren’t going to go and create an API yourself, are you? This is where THREE comes in. THREE is an API for webGL that allows for easy creation of 3d geometry. So, let’s get started:

Script

Everything goes in <script></script> headings when working in THREE.

Get Started

You need three things to display things in THREE:

1. Scene
2. Camera
3. Renderer

var scene = new THREE.Scene();

var camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.1, 1000 );

var renderer = new THREE.WebGLRenderer();

renderer.setSize( window.innerWidth, window.innerHeight ); document.body.appendChild( renderer.domElement );

Camera

There are three types of cameras:

1. Perspective Camera: The first attribute is the Field of View. The second attribute is the aspect ratio. You almost AWLAYS want width of the element divided by height. Next two are near/far clipping.

Renderer

You need to set the size at which to render at. If you render at the width/height of your browser, its crips and clean. If you render it smaller than that, it will look blurry but be less processor intensive.

There are a few renderers available for THREE:

1. WebGLRenderer:
   1. Remember! A WebGLRenderer is required to get transparency, shaders and other such webGL specific functionality running.
2. canvasRenderer: You get much less functionality using a canvas renderer, so be careful about when you decided to use this.

Mesh

A mesh is what the user will see, and it is what is rendered. Alone, a mesh does nothing, it needs two things along with it to work: A geometry and a material. The mesh will take the material and apply it to the geometry and then allow it to be added to the scene.

Some notes on meshes:

* To make a mesh double sided, use the property:

mesh.material.side = THREE.DoubleSide;

Geometry

There are a bunch of geometries in THREE. All of them take their x,y,z size as constructor values.

1. CubeGeometry
2. SphereGeometry
3. PlaneGeometry

Adding to a scene

Use scene.add() to add something to your scene. They start at position 0,0,0, so you might want to move some of them around after adding them.

Rendering the scene

To render a scene, you need a RENDER LOOP.

function render() {

requestAnimationFrame(render);

renderer.render(scene, camera);

}

render();

Moving things

Anything that you want to be moved has to go IN your render loop!

Each mesh has a bunch of methods for moving things, check the mesh docs for what methods there are to move around with.

Rotation

Rotation is calculated in RADIANS. That means 180 degrees is approximately 3.14159 (PI) radians. If you want to rotate something,

Custom Materials

Custom materials are made in the GLSL shading language (openGL). Currently, the easiest way to make one is to embed it in the HTML directly and access it with an innerHTML call. Heres an example:

<script id=*"vertexShader"* type=*"x-shader/x-vertex"*>

varying vec2 vUv;

void main()

{

vUv = uv;

gl\_Position = projectionMatrix \* modelViewMatrix \* vec4( position, 1.0 );

}

</script>

You generally need three things to get a material up and running for a mesh in THREE. A vertex shader, a fragment shader and any associated uniforms. The vertex shader will generally manipulate the objects geometry and the fragment shader will manipulate the texture and its UVs.

Creating a shader is all based on GLSL so look at the notes on that to see how it’s done. The most I will say here is that you ‘call’ the function that ‘creates’ the shader with uniforms. Uniforms, therefore, act as arguments to a function.

Here is an example of creating uniforms:

**this**.customUniforms = {

baseTexture: { type: "t", value: growthTexture },

transitionTexture: { type: "t", value: transitionTexture },

baseSpeed: { type: "f", value: 0.5 },

time: { type: "f", value: 0.0 },

moveX: { type: "f", value: -1.0},

moveY: { type: "f", value: 0.0}

};

The type flag is, as you would think, its type. ‘t’ indicates a sampler2D texture. ‘f’ indicates a float.   
  
After you have created a fragment shader, vertex shader and created the uniforms that will be used to call them, you generate the material with a call such as:

**var** customMaterial = **new** THREE.ShaderMaterial({

uniforms : customUniforms,

vertexShader : document.getElementById('vertexShader').innerHTML,

fragmentShader : document.getElementById('transitionShader').innerHTML

});

Remember, this all applies if you **embed your shader in the html directly.** Which is the easiest method of doing it. It is ugly, and I don’t like it, but for now I do not know of a better way. I will update this section if I find one.