Lab 01: WebGL and THREE.js Configuration

CS423: Computer Graphics

Contents

1 Acquiring THREE.js

So... we want to build applications using `THREE.js` and `WebGL`. The first step in the process is acquiring a copy of the THREE.js. The simple approach is to download a copy of the library from the THREE.js website: `https://github.com/mrdoob/three.js/archive/master.zip`

The downside of this approach is that `THREE.js` is under VERY active development so the approach that I take is to get a clone of their GitHub archive (I'm assuming here that you're working on a computer running Linux). Start by cloning the `THREE.js` repo to your machine:

```
git clone https://github.com/mrdoob/three.js.git
```

The build environment for `THREE.js` uses `Node.js`. You will need to use your package manager of choice to install Node on your build machine. For example, for Debian or Ubuntu-based distributions, you would do the following:

```
sudo apt-get install nodejs npm
```

Do make certain to install the NPM package manager. At this point, you need to do the following:

```
cd ./three.js          # Assuming you're in the folder where
                       # you put the clone
npm install
npm run build
```

This will build two Javascript libraries: `three.js` and `three.min.js` in the `build` folder. The first library is an uncompressed JS file with the source code for the entire library and is recommended for use when debugging. The `three.min.js` library is compressed version of the library used for production. This is a common pattern in `THREE.js` where you have both uncompressed and compressed versions of libraries.

2 Creating A Working Folder

You will need to create a working folder for your pages and programs. For a number of reasons when working in a Linux environment, I prefer to have a web server installed on my development machine with the server configured to serve local user folders. With Apache, this means that a user creates a `public_html` folder in their home folder which is accessed at `http://localhost/~username`. For example, on my machine, the URL is `http://local/ alewis`.

Add a `CS423` subfolder into your `public_html` folder. In this subfolder, add two more folders: `assets` and `libs`. The `assets` folder will be where we will be placing models, pictures, and other things. The `libs` folder will be where we will place libraries such as the `three.js` and `three.min.js` libraries. This folder structure enables the use of relative path names in our HTML.

In the `libs` folder, my practice is to then soft-link to the `three.js` and `three.min.js` libraries in the `THREE.js` build folder.

3 Now The Fun Begins!

Let's start by creating a template that we can use for all of our WebGL webapps. Using your favorite text editor, create the file `skeleton.html` in your working folder. Add the following:

```
<!DOCTYPE html>
<HTML>
  <HEAD>
    <TITLE>CS423 Homework Template</TITLE>
    <SCRIPT
      TYPE="text/javascript"
      SRC="../libs/three.js">
    </SCRIPT>
    <STYLE>
      body {
      margin: 0;
      overflow: hidden;
      }
    </STYLE>
  </HEAD>
  <BODY>
    <DIV ID="WebGL-output">
    </DIV>
    <!-- Scripts that we use for running things -->
    <SCRIPT TYPE="text/javascript">
      // Put the bulk of what we do in the onload handler
      // for the window.
      function init() {
         // Put Three.js stuff here.
      }
      window.onload = init
    </SCRIPT>
  </BODY>
</HTML>
```

Not that we load the `three.js` library from the `libs` folder found in the same location where the script is kept on the web server. The body of the HTML document contains a single `DIV` section that we will use as the container for our `WebGL` output. The associated script defines a function `init()` where we will place the `THREE.js` stuff to draw things. Note how we link the function to the window by setting `onload` handler to be a reference to the function.

Save and load your file into your browser of choice. Check the browser web console to confirm that no errors occurred.

3.1 Let's actually do something

Starting from the template we just built, create a new file named `01-simplescene.html` with the following HTML:

```
<!DOCTYPE html>
<HTML>
<HEAD>
    <TITLE>Example 01.02 - Simple Scene</TITLE>
    <SCRIPT TYPE="text/javascript" SRC="../libs/three.js"></SCRIPT>
    <STYLE>
        body {
            /* set margin to 0 and overflow to hidden, to go fullscreen */
            margin: 0;
            overflow: hidden;
        }
    </STYLE>
</HEAD>
<BODY>

```

```
<DIV id="Stats-output">
</DIV>
<!-- Div which will hold the Output -->
<DIV id="WebGL-output">
</DIV>
<!-- Javascript code that runs our Three.js examples -->
<SCRIPT TYPE="text/javascript" SRC="01-simplescene.js">
</SCRIPT>
</BODY>
</HTML>
```

Note also that we're using a separate file for our Javascript code. There are differing schools of thought about this coding pattern. Some people prefer to do everything buried in script tags in your HTML so that you're managing one file. But others, myself included, will use a separate file for my Javascript to try to keep things better organized.

Let's start building that Javacript file. Create a new file named `01-simplescene.js` and add the following:

```
//
// File:   01-basic-scene.js
// Author: adam.lewis@athens.edu
// Purpose:
// Demo some of the basics of working with the scenegraph.
// This is an extension of code from the Learning THREE.js textbook
// once everything is loaded, we run our Three.js stuff.
function init() {
    var scene = new THREE.Scene();
    var extent = window.innerWidth / window.innerHeight;
    var camera = new THREE.PerspectiveCamera(45, extent, 0.1, 1000);
    var renderer = new THREE.WebGLRenderer();
    renderer.setClearColor(0xEEEEEE,1.0);
    renderer.setSize(window.innerWidth,window.innerHeight);
}
window.onload = init;
```

This gets things started as we did previously by defining a function `init()` and then setting the window `onload` handler to that function. We'll do all of the work within that function. Note also that we will implementing everything as inner functions with `init()`. There's a bunch of `THREE.js` stuff here but roll with me as we'll explain the details over the next few class sessions. In this case, you see the things you have to do in every `THREE.js`-based `WebGL` application to get things started: (1) create a scene, create a camera, and then build a renderer to draw the stuff we want to draw.

Stuff, you say? What sort of stuff. Let's an axis set into the scene to get a frame of reference

```
// Drop a axis set into the scene
var axes = new THREE.AxisHelper(20);
scene.add(axes);
```

Next we will add a base plane upon which we will be setting objects.

```
// Let's add a base plane upon which we place objects.
var planeGeometry = new THREE.PlaneGeometry(60,20);
var planeMaterial = new THREE.MeshBasicMaterial({color:0xCCCCCC});
var plane = new THREE.Mesh(planeGeometry, planeMaterial);
plane.rotation.x = -0.5 * Math.PI;
plane.position.x = 15;
plane.position.y = 0;
plane.position.z = 0;
scene.add(plane);
```

Again, lot's of stuff that makes you say "Wut?". But the workflow is more important here: (1) create a configure a drawing object, (2) position that drawing object in 3d space, and (3) add it to the scene to be rendered. Note how we connect the `WebGL` renderer to the matching HTML element in the DOM.

Let's add some objects into the plane. Note how the pattern flows between creating a geometry object, building a material that will wrap that object, and creating the skeleton of the object using a mesh:

```
// Add a cube
var cubeGeometry = new THREE.BoxGeometry(4,4,4);
var cubeMaterial = new THREE.MeshBasicMaterial({color: 0xFF0000, wireframe: true});
var cube = new THREE.Mesh(cubeGeometry,cubeMaterial);
cube.position.x = -4;
cube.position.y = 3;
cube.position.z = 0;
scene.add(cube);
// And a sphere
var sphereGeometry = new THREE.SphereGeometry(4, 20, 20);
var sphereMaterial = new THREE.MeshBasicMaterial({color: 0x7777ff, wireframe: true});
var sphere = new THREE.Mesh(sphereGeometry,sphereMaterial);
sphere.position.x = 20;
sphere.position.y = 4;
sphere.position.z = 2;
scene.add(sphere);
```

Now we can set the view camera position and render the scene:

```
// Need to tell Three.js the point from where we're viewing the scene
camera.position.x = -30;
camera.position.y = 40;
camera.position.z = 30;
camera.lookAt(scene.position);
// Now update the page by attaching the renderer to appropriate place in the
// HTML DOM for a page and then tell the renderer to render the scene
document.getElementById("WebGL-output").appendChild(renderer.domElement);
renderer.render(scene,camera);
```

Save both the HTML and JS files and load them into your web browser.

4 Submission instructions

Please create a PDF file with the following:

- A screen-shot of both your webapps displayed in the browser.
- HTML and JS files for each webapp

Attach this PDF file to the submission link in Blackboard.