three.js

three.js

- an open source object-oriented JavaScript library for 3D graphics
- provides a scene graph API and renderer(s)
- built on WebGL

Web Page Structure

- web page structure is the same as for WebGL
  - need a canvas

```
<html>

<head>
    <title>Hello three.js!</title>


</head>

<body>
    <h2>Hello three.js!</h2>

    <div id="canvas-holder">
        <canvas id="webglcanvas" width="600" height="600"></canvas>
    </div>
</body>

</html>
```


Basic Template, Part 1

import map defines where to find modules used

import modules used

`as THREE` means that `THREE.` will prefix all elements from that module

`from "three"` refers to the import map

`OrbitControls` allows mouse trackball

names of variables and functions are user-defined, not dictated by three.js

```
<script type="importmap">
{
  "imports": {
    "three": "../threejs/three.module.min.js",
    "addons/": "../threejs/"
  }
}
</script>

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

  let scene, camera, renderer;  // Three.js rendering basics.

  let canvas;  // The canvas on which the image is rendered.

  // initialize when page is loaded
  window.onload = function () { init(); };

  /**
   *  Create the scene graph.  This function is called once, as soon as
   *  the page loads.  The renderer has already been created before this
   *  function is called.
   */
  function createWorld() {
      // set background clear color
      // create scene object
      // create camera
      // create some lights and add them to the scene
      // create the geometry and add it to the scene
  }
```

← calls `init()` when page is loaded

Basic Template, Part 2

webglcanvas and canvas-holder refer to the HTML elements

sets up mouse trackball

```
    /**
     *   Render the scene.  This is called for each frame of the animation.
     */
    function render() {
        renderer.render(scene, camera);
    }

    /**
     *  This init() function is called when by the onload event when the document has loaded.
     */
    function init() {
        try {
            canvas = document.getElementById("webglcanvas");
            renderer = new THREE.WebGLRenderer({
                canvas: webglcanvas,
                antialias: true
            });
        }
        catch (e) {
            document.getElementById("canvas-holder").innerHTML =
                "<h3><b>Sorry, WebGL is required but is not available.</b><h3>";
            return;
        }
        createWorld();

        let controls = new OrbitControls(camera, canvas);
        controls.enablePan = false;
        controls.enableZoom = false;
        controls.addEventListener("change", render);

        render();
    }
</script>
```


Core Elements

`let scene, camera, renderer;`
- typically have global variables for each element

- scene
  - represents the scene graph
- camera
- renderer
  - creates an image from a scene graph

Scene

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

- made up of objects of type `THREE.Object3D` (and subclasses)
  - an object has a list of child nodes
- the scene graph must be a tree
  - clone a node to create a copy of a subgraph

Camera

note reversed order compared to WebGL

```
camera =
  new THREE.OrthographicCamera(left,right,
                               top,bottom,
                               near,far);
```

- defines view volume (EC)
- `near`, `far` are distances from the camera (not z coordinates)
  - `near` < 0 is allowed

```
camera =
  new THREE.PerspectiveCamera(fieldOfViewAngle,
                              aspect,near,far);
```

- `fieldOfViewAngle` is the vertical extent of the view volume, in degrees
- `aspect` ratio is typically set based on the canvas dimensions
  - `canvas.width/canvas.height`
- `near`, `far` are distances from the camera (not z coordinates)
  - 0 < `near` < `far`

Camera Setup

- positioning
  - `camera.position` specifies position

```
camera.position.set(x,y,z);
```

  - set all components at once

```
camera.position.x = x;
camera.position.y = y;
camera.position.z = z;
```

  - set an individual component's value

Camera Setup

- orientation
  - `camera.rotation` specifies rotation angles (in radians) around x, y, z axes respectively
  - rotations are applied in that order – first x rotation, then y, then z
  - `obj.lookAt(vec)`
    - `vec` must be a `Vector3` in OC
      - if `obj` has no parent, OC = WC
    - rotates `obj` so that it faces `vec`, with up direction `obj.up`
      - default for `obj.up` is (0,1,0)

Camera Setup Example

```
// create a camera, sitting on the positive z-axis.  The camera is
// not (yet) part of the scene.
camera = new THREE.PerspectiveCamera(45,
    canvas.width / canvas.height, 1, 30);
camera.position.z = 15;
```


Renderer

```
renderer =
  new THREE.WebGLRenderer( { canvas: canvas,
                             antialias: true } );
```

  - renders using WebGL
    - other renderers are possible, though may not support all of the features of the WebGL renderer
  - *canvas* is the canvas element in the HTML document
  - typically want antialiasing, but `false` is possible

```
renderer.setClearColor(0xhexvalue);
renderer.setClearColor("rgb(r,g,b)");
```

  - set the background clear color
    - r,g,b values are 0-255

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

  - render scene

Modeling – Objects

- five kinds of basic objects – subclasses of `THREE.Object3D`
  - `THREE.Points` – corresponds to `GL_POINTS`
  - `THREE.LineSegments` – corresponds to `GL_LINES`
  - `THREE.Line` – corresponds to `GL_LINE_STRIP`
  - `THREE.LineLoop` – corresponds to `GL_LINE_LOOP`
  - `THREE.Mesh` – corresponds to `GL_TRIANGLES`
- object consists of geometry + material

  `let obj = new THREE.Mesh(`*`geometry`*`,`*`material`*`);`
  - definitions for common mesh geometries are provided

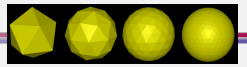
Modeling – Geometry

- defining geometry
  - there are definitions for common mesh geometries
  - in JavaScript, can omit values for later parameters

```
new THREE.CylinderGeometry(radiusTop, radiusBottom, height,
          radiusSegments, heightSegments, openEnded, thetaStart, thetaLength)
```

```
new THREE.BoxGeometry(width, height, depth,
                        widthSegments, heightSegments, depthSegments)

new THREE.PlaneGeometry(width, height, widthSegments, heightSegments)

new THREE.RingGeometry(innerRadius, outerRadius, thetaSegments, phiSegments,
                        thetaStart, thetaLength)

new THREE.ConeGeometry(radiusBottom, height, radiusSegments,
                        heightSegments, openEnded, thetaStart, thetaLength)

new THREE.SphereGeometry(radius, widthSegments, heightSegments,
                        phiStart, phiLength, thetaStart, thetaLength)

new THREE.TorusGeometry(radius, tube, radialSegments, tubularSegments, arc)
```

also `TetrahedronGeometry`, `OctahedronGeometry`, `DodecahedronGeometry`, `IcosahedronGeometry`
with parameters `size` and `detail`



Modeling – Materials

- for points objects, `THREE.PointsMaterial`
  - has properties `color`, `size`, `sizeAttenuation`
- for lines, `THREE.LineBasicMaterial`
  - has properties `color`, `linewidth`
  - see reading for how to specify different colors for each vertex
- for meshes
  - `THREE.MeshBasicMaterial` – fixed color, not affected by lighting
    - has property `color`
  - `THREE.MeshLambertMaterial` – emission and diffuse only
    - has properties `color`, `emissive`
  - `THREE.MeshPhongMaterial` – includes specular
    - has properties `color`, `emissive`, `specular`, `shininess`

Object Creation Example

```
// create the geometry
let donut = new THREE.Mesh(
    new THREE.TorusGeometry(2, .5, 16, 32),
    new THREE.MeshLambertMaterial({
        color: "rgb(255,128,0)"
    }));


// don't forget to add to the scene!
scene.add(donut);
```

Modeling – Materials

- additional properties for mesh materials

- `wireframe` — a boolean value that indicates whether the mesh should be drawn as a wireframe model, showing only the outlines of its faces. The default is *false*. A *true* value works best with *MeshBasicMaterial*.
- `wireframeLinewidth` — the width of the lines used to draw the wireframe, in pixels. The default is 1. (Non-default values might not be respected.)
- `visible` — a boolean value that controls whether the object on which it is used is rendered or not, with a default of *true*.
- `side` — has value *THREE.FrontSide*, *THREE.BackSide*, or *THREE.DoubleSide*, with the default being *THREE.FrontSide*. This determines whether faces of the mesh are drawn or not, depending on which side of the face is visible. With the default value, *THREE.FrontSide*, a face is drawn only if it is being viewed from the front. *THREE.DoubleSide* will draw it whether it is viewed from the front or from the back, and *THREE.BackSide* only if it is viewed from the back. For closed objects, such as a cube or a complete sphere, the default value makes sense, at least as long as the viewer is outside of the object. For a plane, an open tube, or a partial sphere, the value should be set to *THREE.DoubleSide*. Otherwise, parts of the object that should be in view won't be drawn.
- `flatShading` — a *boolean* value, with the default being *false*. This does not work for *MeshBasicMaterial*. For an object that is supposed to look "faceted," with flat sides, it is important to set this property to *true*. That would be the case, for example, for a cube or for a cylinder with a small number of sides.

  - `polygonOffset`, `polygonOffsetUnits`, `polygonOffsetFactor`
    - defined as material properties (apply to the solid object so it doesn't interfere with the wireframe version)

Colors

- `THREE.Color` represents an RGB color
  - has properties `r`, `g`, `b` with values in the range 0-1
- can specify values using color names, RGB values, RGB strings, hexadecimal values

```
var c1 = new THREE.Color("skyblue");
var c2 = new THREE.Color(1,1,0);  // yellow
var c3 = new THREE.Color(0x98fb98);  // pale green
var c4 = new THREE.Color("rgb(255,128,0)");
```

- can often use just color name or value instead of a `THREE.Color` object in places where a color is expected
  - e.g. in material properties

```
let donut = new THREE.Mesh(
    new THREE.TorusGeometry(2, .5, 16, 32),
    new THREE.MeshLambertMaterial({
        color: "rgb(255,128,0)"
    }));
```


Modeling Transformations

- `THREE.Object3D` properties
  - `scale` (type `THREE.Vector3`)
    - has properties `x`, `y`, `z` specifying scale factor in each direction
  - `rotation` (type `THREE.Euler`)
    - has properties `x`, `y`, `z` specifying rotations around the respective axes
      - applied in that order
  - `position` (type `THREE.Vector3`)
    - has properties `x`, `y`, `z` specifying translation amounts
  - applied in the order scale, rotation, translation
  - `x`, `y`, `z` can be set individually or all at once
    - e.g. `obj.scale.set(2,2,2);`
    - e.g. `obj.scale.y = 0.5;`

Modeling Transform Example

```
// create the geometry
let donut = new THREE.Mesh(
    new THREE.TorusGeometry(2, .5, 16, 32),
    new THREE.MeshLambertMaterial({
        color: "rgb(255,128,0)"
    }));
donut.rotation.set(-Math.PI / 4, Math.PI / 4, 0);
```

Hierarchical Modeling

- build subgraphs to represent complex objects

```
let tree = new THREE.Object3D();
let trunk = new THREE.Mesh(
    new THREE.CylinderGeometry(0.2,0.2,1,16,1),
    new THREE.MeshLambertMaterial({
        color: 0x885522
    })
);
trunk.position.y = 0.5;  // move base up to origin
let leaves = new THREE.Mesh(
    new THREE.ConeGeometry(.7,2,16,3),
    new THREE.MeshPhongMaterial({
        color: 0x00DD00,
        specular: 0x006000,
        shininess: 5
    })
);
leaves.position.y = 2;  // move bottom of cone to top of trunk
tree.add(trunk);
tree.add(leaves);
```

```
tree.position.set(-1.5,0,2);
tree.scale.set(0.7,0.7,0.7);
diskworldModel.add(tree.clone());
tree.position.set(-1,0,5.2);
tree.scale.set(0.25,0.25,0.25);
diskworldModel.add(tree.clone());
tree.position.set(5.1,0,0.5);
tree.scale.set(0.3,0.3,0.3);
diskworldModel.add(tree.clone());
tree.position.set(5.1,0,-0.5);
tree.scale.set(0.35,0.35,0.35);
diskworldModel.add(tree.clone());
tree.position.set(5.3,0,0);
tree.scale.set(0.5,0.5,0.5);
diskworldModel.add(tree.clone());
tree.position.set(-3.6,0,-3.6);
tree.scale.set(0.6,0.6,0.6);
diskworldModel.add(tree.clone());
tree.position.set(1,0,-5);
tree.scale.set(0.35,0.35,0.35);
diskworldModel.add(tree.clone());
```

the clone gets its own copy of the object's properties so changing the properties in the original object doesn't affect the clone