Building a 3D scene

3D content is a really interesting way to create interactive content for the web but until recently it has been a pain to develop on a Mac, particularly since most device makers decided early on that Macintosh hardware wasn't powerful enough to work with their devices (see, this post from 2015 outlining the minimal hardware support level and, partly, why it won't work with Apple hardware) and, though things seem to be getting better with newer Apple hardware releases, I'm not holding my breath and neither is Macworld UK.

While direct development with the Oculus SDK is out of the question we can always fallback into libraries that sit on top of the raw WebGL spec and allow us to use the API without having to worry about Shaders and Matrix algebra.

We'll look at the basics of two frameworks/libraries:

- Three.js one of the best 3D and VR frameworks available
- A-Frame, a project from Mozilla that uses declarative markup to create 3D content

Finally we'll look at the WebXR Device API specification as a unified toolkit for 3D immersive media on the web along with some ideas that are soon to hit the prototype stage soon.

Three.js

Three.js sits on top of WebGL and provides abstractions to most WebGL primitives and the ability to drop to the low level shaders and functions when needed.

I'm further developing the full example in Glitch where you're more than welcome to remix it for your own needs.

The code

The first part of the code sets up basic variables to setup the environment for the scene.

THREE.Scene() defines the environment that will bbe rendered. This is the root of the Three.js application.

THREE.PerspectiveCamera defines how the scene will be rendered. The camera takes 4 parameters

- `fov` — Camera frustum vertical field of view.
- `aspect` — Camera frustum aspect ratio.
- `near` — Camera frustum near plane.
- `far` — Camera frustum far plane.

These four parameters together define the region of the 3D image that appears on the screen (the viewing fustrum)

THREE.WebGLRenderer tells Thre.js to use WebGL to render the scene. There are other renderers available.

We set the size of the scene to be the full height and width of the screen. We use innerWidth/innerHeight to make sure we take into account the dimensions of the browser's chrome.

Finally we attach the renderer to the page using append`Child`.

```
const scene = new THREE.Scene();
const camera = new THREE.PerspectiveCamera(75,
  window.innerWidth / window.innerHeight,
  0.1,
  1000);
const renderer = new THREE.WebGLRenderer();
renderer.setSize(window.innerWidth, window.innerHeight);
document.body.appendChild(renderer.domElement);
```

At the most basic level, objects in Three.js are made of two items: a geometry and a material.

The geometry tells Three.js what shape the object has: a sphere or a plane in this example. We then add each individual object to the scene by calling `scene.`add with the new object we want to add as parameter.

```
const geometry = new THREE.SphereGeometry( 2, 32, 32 );
const material = new THREE.MeshBasicMaterial({
  color: 0x0000ff,
  wireframe: true
});
const sphere = new THREE.Mesh(geometry, material);
scene.add(sphere);

const geo = new THREE.PlaneGeometry(36, 36, 36);
const mat = new THREE.MeshBasicMaterial({
  color: 0xff00ff,
  side: THREE.DoubleSide });
const plane = new THREE.Mesh(geo, mat);
scene.add(plane);
```

We next set the position of the objects. In this caase we set the sphere and the plane on an X, Y, Z coordinate sysem and then rotate the plane 30 units so it looks flatter than the sphere.

```
sphere.position.set(0, 1 , 0);
plane.position.set( -1, -1, 0);
plane.rotateX( 30 );
```

The `render` function is where the animations happen. We call requestAnimationFrame() to animate the next frame of animation. Then we change the rotation of the x and y axes and render the scene with the camera we defined earlier.

The last step is to call the `rendeer()` function to kick off the animation for the first time.

```
const render = () => {
  requestAnimationFrame(render);
  sphere.rotation.x += 0.01;
  sphere.rotation.y += 0.01;
  renderer.render(scene, camera);
}
render();
```

A-Frame

A frame sits on top of Three.js with a declarative API based on tags, like HTML, to declare the components of a 3D scene. Another thing that A-Frame does that Three.JS doesn't do by default is provide a VR-Ready experience for the content we create without writing Javascript.

Some things that I've had to keep in mind while working with A-Frame:

A-Frame uses a different system to position content. It uses a right-handed coordinate system where the negative Z axis extends into the screen.

Nested elements and their positioning in their relation to their parents.

See the position attribute that is common to all A-Frame elements.

Because A-Frame extends Three.js it's important to understand the relationship between the two. In particular:

- A-Frame's `<a-scene>` maps one-to-one with a three.js scene.
- A-Frame's `<a-entity>` maps to one or more three.js objects.
- three.js's objects have a reference to their A-Frame entity via `.el`, which is set by A-Frame.

This example creates a similar scene than the one we created with Three.js.

Note how the elements are nested, the sphere, the plane, and a sky are nested inside the scene element; and the animation for the sphere is nested inside the sphere that is being animated.

```
<a-scene>
  <a-sphere
      position="0 2 -4"
      radius="1.25"
      color="teal"
      wireframe="true"
      rotation="0 90 0">

      <a-animation
```

```
          attribute="rotation"
          dur="500"
          from="0 0 0"
          to="0 360 0"
          repeat="indefinite"></a-animation>
    </a-sphere>

    <a-plane
      position="0 0 -4"
      rotation="-90 0 0"
      width="12" height="12"
      color="#7BC8A4"></a-plane>

    <a-sky color="#ECECEC"></a-sky>
  </a-scene>
```

Other APIs: WebXR

The final API that I want to discuss in this post is WebXR a new API that handles both VR and AR using the same API.

What I found most intriguing about WebXR is the possibility of creating “magic windows” that will work based on the device’s capabilities. If the device doesn’t have a 3D-enable viewer attached to it then nothing happens and you see a 2D image of the experience.

If the device has a 3D viewer attached then the user can select to go full VR or AR on the experience. This is particularly cool when working with AR because I can then place and interact with the object anywhere around me.

Chrome continues to run an Origin Trial for WebXR to gather feedback. You can enable the feature on individual versions of Chrome by enabling the “Experimental Web Features” flag in settings.

The purpose of the trial is to gather feedback from developers as they work on finalizing the API to push standardization.