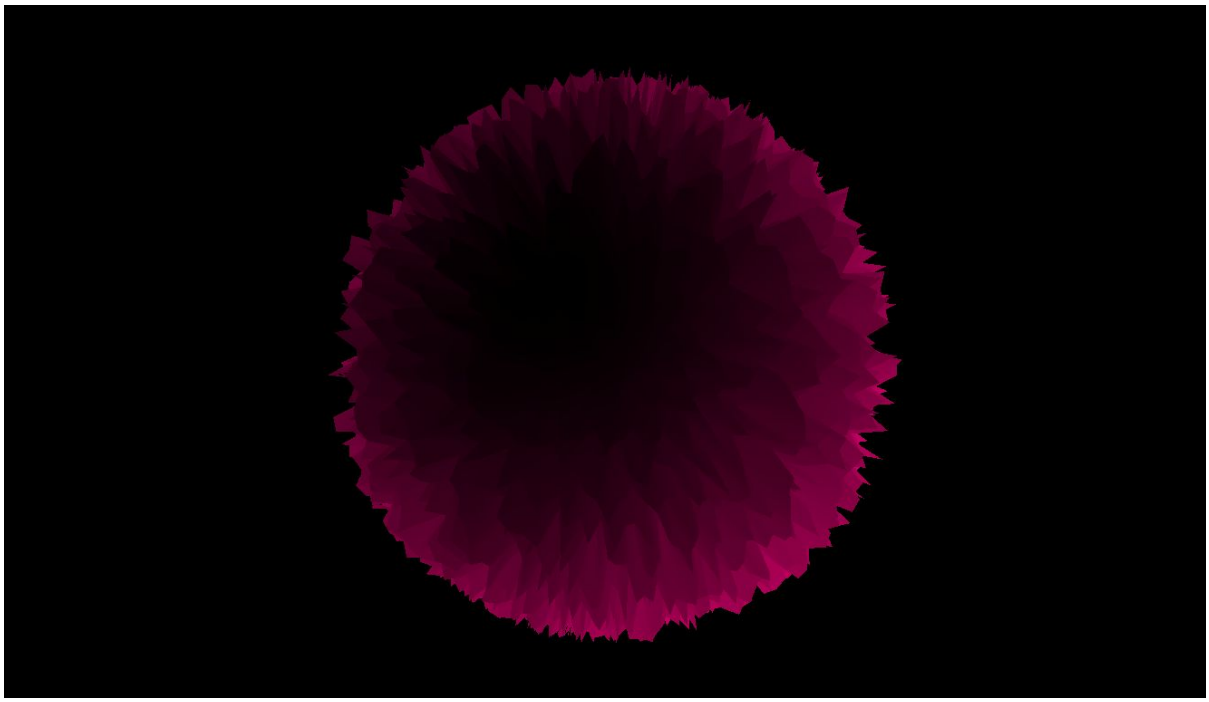
50.017 Graphics and Visualization

WebGL Lab Session 2: Using Custom Shader with Three.js



Overview

In this lab session, we will be creating a morphing sphere by writing our own shader program.

GLSL

WebGL shaders are written in GLSL (OpenGL Shader Language), a C-like program that runs in GPU and manipulates the vertex and texture data you create in JavaScript. There are two type of shaders:

Vertex Shaders - operate on the coordinate data contained in the vertex buffers
Fragment Shaders - determine the color of each pixel

Using Custom Shader with Three.js

There's several built-in material provided by threejs, such as phong material, lambert material and etc. These materials have shader program pre-made by

Three.js library. However if we want to create special effects, we have to create our own shader program. To use custom shader in Three.js, you have to use “THREE.ShaderMaterial”, which you should supply vertex shader and fragment shader when creating the material.

```
var material = new THREE.ShaderMaterial( {

	uniforms: {
		time: { type: "f", value: 1.0 },
		resolution: { type: "v2", value: new THREE.Vector2() }
	},
	vertexShader: document.getElementById( 'vertexShader' ).textContent,
	fragmentShader: document.getElementById( 'fragmentShader' ).textContent

} );
```

For more details regarding to shader material, please refer to:
http://threejs.org/docs/#Reference/Materials/ShaderMaterial

Project

Please download the starter code to start your project, below is a step by step tutorial to create a basic shader program and use with your webgl program.

1. In the HTML file, you should see three <script> tag, each with different id. The script with ‘main’ id should be remain untouch. In this project, you should mainly modify this two part, which are the script with ids: “vertex-shader” and “fragment-shader”.

2. In the ‘main’ script, there’s a shader material used for the sphere, which we should supply with our own shader program.

3. modify the vertex-shader script

```
<script type="x-shader/x-vertex" id="fragmentShader">
  // Put the Fragment Shader code here
  varying vec3 fNormal;

  uniform vec3 color;

  void main() {

    vec3 light = vec3( 0.5, 0.2, 1.0 );
```

```
        light = normalize( light );

        float dProd = dot( fNormal, light ) * 0.5 + 0.5;

        // colour is RGBA: u, v, 0, 1
        gl_FragColor = vec4( vec3( dProd ) * vec3( color ), 1.0 );

    }

</script>
```

And in the window onload callback function, add the following lines:

```
        uniforms = {

         amplitude: { type: "f", value: 1.0 },

        };

        displacement = new Float32Array( sphereGeometry.attributes.position.count );
        noise = new Float32Array( sphereGeometry.attributes.position.count );

        for (var i = 0; i < displacement.length; i++) {

            noise[ i ] = Math.random() * 5;

        }

        sphereGeometry.addAttribute( 'displacement', new THREE.BufferAttribute( noise, 1 ) );

        material = new THREE.ShaderMaterial( {
            uniforms: uniforms,
            vertexShader: document.getElementById( 'vertexShader' ).textContent,
            fragmentShader: document.getElementById( 'fragmentShader' ).textContent
        } );
```

4. In the fragment-shader script

```
<script type="x-shader/x-vertex" id="fragmentShader">
    // Put the Fragment Shader code here
    varying vec3 fNormal;

    uniform vec3 color;

    void main() {

        vec3 light = vec3( 0.5, 0.2, 1.0 );
        light = normalize( light );
```

```
        float dProd = dot( fNormal, light ) * 0.5 + 0.5;

        // colour is RGBA: u, v, 0, 1
        gl_FragColor = vec4( vec3( dProd ) * vec3( color ), 1.0 );

    }

</script>
```

And in the window onload callback function, add the color attributes to the uniforms:

```
        uniforms = {

            amplitude: { type: "f", value: 1.0 },
            color: { type: "c", value: new THREE.Color( 0xff2200 ) }

        };
```

5. To morph the object we will pass in the time attribute to our shader

In the render() function, add the following lines above the render function call

```
        var time = Date.now() * 0.01;
        sphere.rotation.y = sphere.rotation.z = 0.01 * time;
        uniforms.amplitude.value = 2.5 * Math.sin( sphere.rotation.y * 0.125);

        // Adds given , s, and l to this color's existing h, s, and l values.
        uniforms.color.value.offsetHSL( 0.0005, 0, 0 );

        for (var i = 0; i < displacement.length; i++) {
            displacement[i] = Math.sin( 0.1 * i + time );

            noise[ i ] += 0.5 * ( 0.5 - Math.random() );
            noise[ i ] = THREE.Math.clamp( noise[ i ], -5, 5 );

            displacement[ i ] += noise[ i ];

        }

        sphere.geometry.attributes.displacement.needsUpdate = true;
```

Appendix

Serving shaders program to your web application

As glsl is a C-like program, one cannot directly pass glsl program to your webgl application. It should be passed as string instead, below are 4 possible options to pass it (but not limited to these 4 methods):

Option 1 - embed as HTML content, which is used in this lab session:

```
<script type="x-shader/x-vertex" id="vertexShader">
    // Put the Vertex Shader code here
</script>
```

Option 2 - as a javascript string

```
var vertexShader = [ "asdf",
                     "asdf",
                     "asdf"].join("\n");
```

Option 3 - also javascript string, but following ES6 standard

```
var vertexShader = `// your vertexShader
    // here in multiline
    `;
```

Option 4 - using xhr request / ajax to get the program as a normal text file and pass the content as string.