Creating your own shaders with the ShaderMaterial

The `THREE.ShaderMaterial` is one of the most versatile and complex materials available in the Three.js library. With it, you can pass in your own custom shaders that are directly run in the WebGL context. A shader is what converts the Three.js JavaScript objects into pixels on the screen. With these custom shaders, you can define exactly how your object should be rendered and overridden, or alter the defaults from the Three.js library. In this section we won't go into the details of how to write custom shaders yet; for more information on that, see *Chapter 11*, *Custom Shaders and Render Post Processing*. For now we'll just look at a very basic example that shows how you can configure this material.

The `ShaderMaterial` has a number of properties that you can set; the ones that we've already seen are as follows:

Name	Description
`wireframe`	This property renders the material as a wireframe. It is great for debugging purposes.
`wireframeLinewidth`	If you enable the `wireframe` property, this property defines the width of the wires from the wireframe.
`shading`	This defines how shading is applied. The possible values are `THREE.SmoothShading` and `THREE.FlatShading`. This property isn't enabled in the example for this material. For an example, look at the section on the `MeshNormalMaterial`.
`vertexColors`	You can define individual colors to be applied to each vertex with this property. This property doesn't work on the `CanvasRenderer`, but it works on the `WebGLRenderer`. For an example, look at the `LineBasicMaterial` example, where we will use this property to color the various parts of a line.
`fog`	This defines whether the material is affected by the global fog settings. It is not shown in action, but if set to `false` the global `fog` property that we saw in *Chapter 2*, *Working with the Basic Components That Make Up a Three.js Scene*, doesn't affect how this object is rendered.

Besides these properties that we've already discussed in previous sections, the `ShaderMaterial` has a number of specific properties that you can use to pass in and configure your custom shader. They may seem a bit obscure at the moment; for more details, see *Chapter 9*, *Animations and Moving the Camera*.

Name	Description
`fragmentShader`	This shader defines the color of each pixel that is passed in.
`vertexShader`	This shader allows you to change the position of each vertex that is passed in.
`uniforms`	This allows you to send information to your shader. The same information is sent to each vertex and fragment.
`defines`	The value of this property is converted to `#define` code in the `vertexShader` and `fragmentShader`. This property can be used to set some global variables in the shader programs.
`attributes`	This can change between each vertex and fragment. Usually used to pass positional and normal-related data. If you want to use this, you need to provide information for all the vertices of the geometry.
`lights`	This defines whether light data should be passed into the shaders. Defaults to `false`.

Before we look at an example, here's a quick explanation about the most important parts of the `ShaderMaterial`: to work with this material, we have to pass in two different shaders:

- `vertexShader`: The `vertexShader` is run on each vertex of the geometry. You can use this shader to transform the geometry by moving the position of the vertices around.
- `fragmentShader`: The `fragmentShader` is run on each pixel of the geometry. In the `vertexShader`, we will return the color that should be shown for this specific pixel.

For all the materials that we've discussed so far in this chapter, the Three.js library provides its own `fragmentShader` and `vertexShader`, so you don't have to worry about it.

For this section we'll look at a simple example that uses a very simple `vertexShader` that changes the x, y, and z coordinates of the vertices of a cube, and a `fragmentShader` that uses the shaders from `glsl.heroku.com` to create an animating material.

Up next you can see the complete code for the `vertexShader` that we'll use.

Writing shaders isn't done in JavaScript. You have to write shaders in a C-like language called GLSL.

```
<script id="vertex-shader" type="x-shader/x-vertex">
    uniform float time;

    void main()
    {
    vec3 posChanged = position;
    posChanged.x = posChanged.x*(abs(sin(time*1.0)));
    posChanged.y = posChanged.y*(abs(cos(time*1.0)));
    posChanged.z = posChanged.z*(abs(sin(time*1.0)));

    gl_Position = projectionMatrix
                  * modelViewMatrix
                  * vec4(posChanged,1.0);
    }
</script>
```

We won't go into too much detail here, and just focus on the most important parts of this code snippet. To communicate with the shaders from JavaScript, we will use something called `uniforms`. In this example we will use the `uniform float time;` statement to pass in an external value. Based on this value, we will change the x, y, and z coordinates of the passed in vertex (which is passed in as the `position` variable):

```
vec3 posChanged = position;
posChanged.x = posChanged.x*(abs(sin(time*1.0)));
posChanged.y = posChanged.y*(abs(cos(time*1.0)));
posChanged.z = posChanged.z*(abs(sin(time*1.0)));
```

The `posChanged` vector now contains the new coordinates for this vertex, based on the passed in `time` variable. The last step that we need to do is pass this new position back to the Three.js library, which is always done as shown:

```
gl_Position = projectionMatrix * modelViewMatrix
              * vec4(posChanged,1.0);
```

The `gl_Position` is a special variable that is used to return the final position.

Next we need to create a `shaderMaterial` and pass in the `vertexShader`. For this we've created a simple helper function:

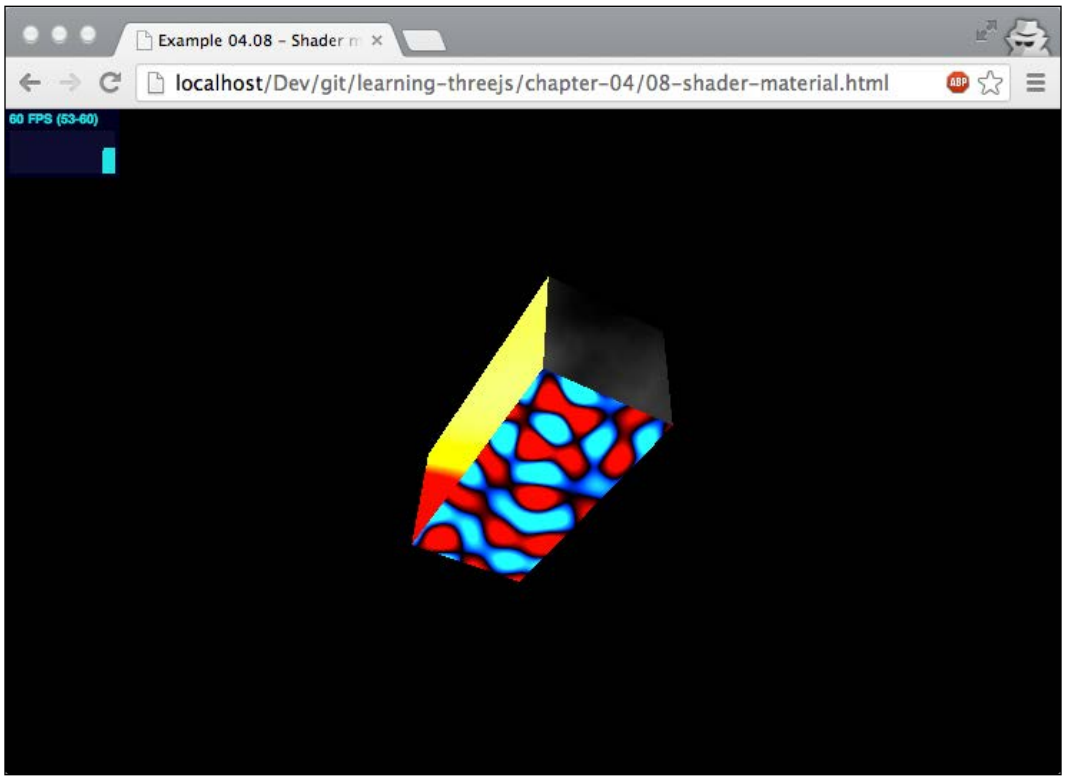
```
function createMaterial(vertexShader, fragmentShader) {
    var vertShader =
            document.getElementById(vertexShader).innerHTML;
    var fragShader =
            document.getElementById(fragmentShader).innerHTML;

    var attributes = {};
    var uniforms = {
        time: {type: 'f', value: 0.2},
        scale: {type: 'f', value: 0.2},
        alpha: {type: 'f', value: 0.6},
        resolution: { type: "v2", value: new THREE.Vector2() }
    };

    uniforms.resolution.value.x = window.innerWidth;
    uniforms.resolution.value.y = window.innerHeight;

    var meshMaterial = new THREE.ShaderMaterial({
        uniforms: uniforms,
        attributes: attributes,
        vertexShader: vertShader,
        fragmentShader: fragShader,
        transparent: true

    });
    return meshMaterial;
}
```

The function that we have created is used as shown: `var meshMaterial1 = createMaterial("vertex-shader","fragment-shader-1");`. The arguments point to the ID of the script element in the HTML page. Here you can also see that we have set up a `uniforms` variable. This variable is used to pass information from our `renderer` into our shader. The complete `render` loop for this example is shown as follows:

```
function render() {
    stats.update();

    cube.rotation.y = step += 0.01;
    cube.rotation.x = step;
```

```
        cube.rotation.z = step;

        cube.material.materials.forEach(function (e) {
            e.uniforms.time.value += 0.01;
        });

        // render using requestAnimationFrame
        requestAnimationFrame(render);
        renderer.render(scene, camera);
    }
```

You can see that we have increased the `time` variable by `0.01` each time the `render` loop is run. This information is passed to our `vertexShader` and is used to calculate the new position of the vertices of our `cube`. Now open the `07-shader-material.html` example and you'll see that the cube shrinks and grows around its axis, as shown in the following screenshot:



In this example you can see that each `cube` face has an animating pattern. The `fragmentShader` that is assigned to each face of the `cube` creates these patterns. As you might have guessed, we've used the `MeshFaceMaterial` for this, as shown in the following code snippet:

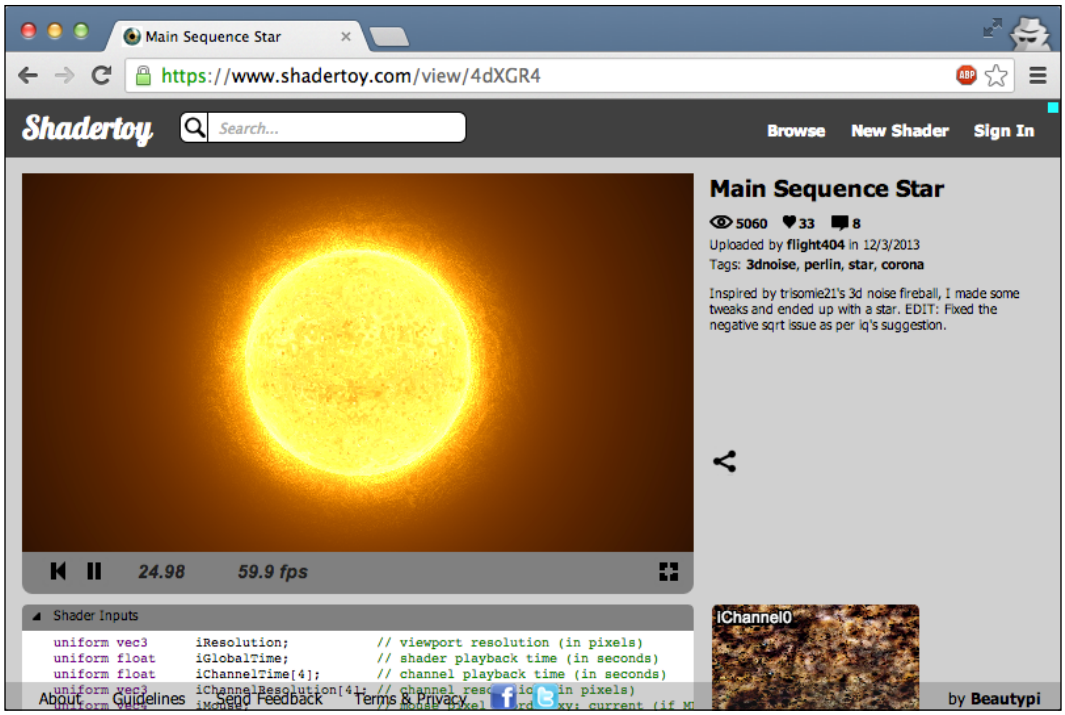
```
var cubeGeometry = new THREE.CubeGeometry(20, 20, 20);

var meshMaterial1 = createMaterial("vertex-shader", "fragment-
shader-1");
var meshMaterial2 = createMaterial("vertex-shader",
                                   "fragment-shader-2");
var meshMaterial3 = createMaterial("vertex-shader",
                                   "fragment-shader-3");
var meshMaterial4 = createMaterial("vertex-shader",
                                   "fragment-shader-4");
var meshMaterial5 = createMaterial("vertex-shader",
                                   "fragment-shader-5");
var meshMaterial6 = createMaterial("vertex-shader",
                                   "fragment-shader-6");

var material = new THREE.MeshFaceMaterial([meshMaterial1,
                   meshMaterial2, meshMaterial3, meshMaterial4,
                   meshMaterial5, meshMaterial6]);

var cube = new THREE.Mesh(cubeGeometry, material);
```

The only part that we haven't explained yet is the `fragmentShader`. For this example, all the fragment shaders were copied from `http://glsl.heroku.com`. This site provides an experimental playground where you can write and share fragment shaders. I won't go into detail here, but the `fragment-shader-6` used in this example looks like the code snippet that follows:

```
<script id="fragment-shader-6" type="x-shader/x-fragment">
    #ifdef GL_ES
    precision mediump float;
    #endif

    uniform float time;
    uniform vec2 resolution;


    void main( void )
```

```
        {

        vec2 uPos = ( gl_FragCoord.xy / resolution.xy );

        uPos.x -= 1.0;
        uPos.y -= 0.5;

        vec3 color = vec3(0.0);
        float vertColor = 2.0;
        for( float i = 0.0; i < 15.0; ++i ) {
        float t = time * (0.9);

        uPos.y += sin( uPos.x*i + t+i/2.0 ) * 0.1;
        float fTemp = abs(1.0 / uPos.y / 100.0);
        vertColor += fTemp;
        color += vec3( fTemp*(10.0-i)/10.0
                     ,fTemp*i/10.0, pow(fTemp,1.5)*1.5 );
        }

        vec4 color_final = vec4(color, 1.0);
        gl_FragColor = color_final;
        }
    </script>
```

The color that finally gets passed back to the Three.js library is the one set to `gl_FragColor = color_final;`. A good way to get a bit more feeling for fragment shaders is by exploring what's available at `http://glsl.heroku.com` and to use the code for your own objects. Before we move on to the next material, the following is one more example of what is possible with a custom vertex shader (`https://www.shadertoy.com/view/4dXGR4`):



Much more on the subject of fragment and vertex shaders can be found in *Chapter 11, Custom Shaders and Render Post Processing*.

Using the materials for a line geometry

The last couple of materials that we're going to look at can only be used on one specific geometry: the `THREE.Line`. As the name implies this is a single line that only consists of vertices and doesn't contain any faces. The Three.js library provides two different materials that you can use on a line, as follows:

- `LineBasicMaterial`: The basic material for a line that allows you to set the colors, line width, line cap, and line join properties
- `LineDashedMaterial`: Has the same properties as the `LineBasicMaterial`, but allows you to create a dashed effect by specifying the dash and spacing sizes

We'll start with the basic variant, and after that we'll look at the dashed variant.