CS 411 – Computer Graphics

Assignment 6 – Texture Mapping

Mayank Bansal – mbansal5@hawk.iit.edu

To allow for both the textures to be loaded in the image, I first added this in the main function to load the texture and the normal map as well:

```
// load texture
texture2 = loadTexture(gl, '../data/normal-map.png');
texture = loadTexture(gl, '../data/frac2.png');
```

In the drawScene function, we bind the normal-map so that we can edit the shader program so that we have both textures binded:

```
// Specify the texture map
gl.activeTexture(gl.TEXTURE0); // use texture unit 0
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.uniform1i(program.u_Sampler, 0);


if (bumpMap) {
    // Specify the texture map
    gl.activeTexture(gl.TEXTURE1); // use texture unit 1
    gl.bindTexture(gl.TEXTURE_2D, texture2);
    gl.uniform1i(program.u_NormalSampler, 1);
}
```

To be able to sample the normals, we have to add a uniform variable attribute that we can use in the shader, therefor we add the following line in getShaderVariables:

```
program.u_NormalSampler = gl.getUniformLocation(program, 'u_NormalSampler');
```

So now our function looks like this:

```
function getShaderVariables(program) {
    //get the storage locations of attribute and uniform variables
    program.a_Position = gl.getAttribLocation(program, 'a_Position');
    program.a_Normal = gl.getAttribLocation(program, 'a_Normal');
    program.a_Color = gl.getAttribLocation(program, 'a_Color');
    program.a_Texture = gl.getAttribLocation(program, 'a_Texture');
    program.u_MvpMatrix = gl.getUniformLocation(program, 'u_MvpMatrix');
    program.u_NormalMatrix = gl.getUniformLocation(program, 'u_NormalMatrix');
    program.u_Sampler = gl.getUniformLocation(program, 'u_Sampler');
    program.u_NormalSampler = gl.getUniformLocation(program, 'u_NormalSampler');
```

```
    if (program.a_Position < 0 || program.a_Normal < 0 || program.a_Color < 0 ||
program.a_Texture < 0 ||
        !program.u_MvpMatrix || !program.u_NormalMatrix || !program.u_Sampler ||
!program.u_NormalSampler) {
        console.log('Error getting attribute/uniform location');
        return false;
    }

    return true;
}
```

We then change the fragment shader to to take care of the other normal map. The normal color is taken as the diffuse color, and the normal is calculated by multiplying by 2 and subtracting 1.

```
// fragment shader program
var FSHADER_SOURCE =
    '#ifdef GL_ES\n' +
    'precision mediump float;\n' +
    '#endif\n' +
    'uniform sampler2D u_Sampler;\n' +
    'uniform sampler2D u_NormalSampler;\n' +
    'varying vec4 v_Color;\n' +
    'varying vec2 v_TexCoord;\n' +
    'void main() {\n' +
    '  vec4 DiffuseColor = texture2D(u_Sampler, v_TexCoord);\n' +
    '  vec3 normal = normalize(2.0 * texture2D(u_NormalSampler, v_TexCoord).rgb -
1.0);\n' +
    '  vec3 LightDir = vec3(-0.35, 0.35, 0.87);\n' +
    '  vec3 L = normalize(LightDir);\n' +
    '  vec3 FinalColor = DiffuseColor.rgb * max(dot(normal, L), 0.0);\n' +
    '  gl_FragColor = v_Color * vec4(FinalColor, DiffuseColor.a);\n' +
    '}\n';
```

The light direction is normalized and we the calculate the final color with the diffused color and the dot product value of N and L.

To calculate the TBN matrix, we use the following function:

```
function findTBN() {
    for (var i = 0; i < model.arrays.indices.length / 3; i++) {

        var p0 = {
            x: model.arrays.vertices[3 * model.arrays.indices[3 * i]],
            y: model.arrays.vertices[3 * model.arrays.indices[3 * i] + 1],
            z: model.arrays.vertices[3 * model.arrays.indices[3 * i] + 2]
        };

        var p1 = {
            x: model.arrays.vertices[3 * model.arrays.indices[3 * i + 1]],
            y: model.arrays.vertices[3 * model.arrays.indices[3 * i + 1] + 1],
            z: model.arrays.vertices[3 * model.arrays.indices[3 * i + 1] + 2]
        };

        var p2 = {
            x: model.arrays.vertices[3 * model.arrays.indices[3 * i + 2]],
            y: model.arrays.vertices[3 * model.arrays.indices[3 * i + 2] + 1],
            z: model.arrays.vertices[3 * model.arrays.indices[3 * i + 2] + 2]
        };
```

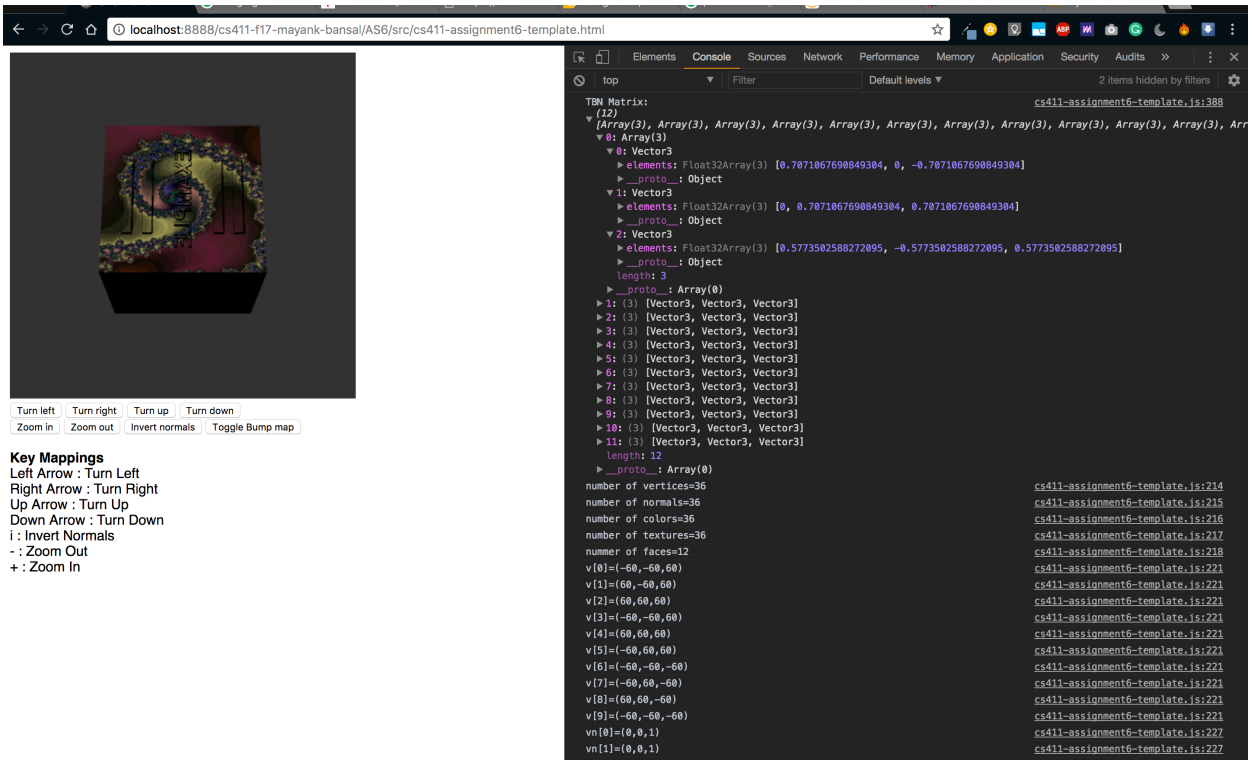
```
        var t0 = {
            x: model.arrays.textures[2 * model.arrays.indices[3 * i]],
            y: model.arrays.textures[2 * model.arrays.indices[3 * i] + 1]
        };

        var t1 = {
            x: model.arrays.textures[2 * model.arrays.indices[3 * i + 1]],
            y: model.arrays.textures[2 * model.arrays.indices[3 * i + 1] + 1]
        };

        var t2 = {
            x: model.arrays.textures[2 * model.arrays.indices[3 * i + 2]],
            y: model.arrays.textures[2 * model.arrays.indices[3 * i + 2] + 1]
        };

        var e1 = {
            x: p1.x - p0.x,
            y: p1.y - p0.y,
            z: p1.y - p0.z
        };

        var e2 = {
            x: p2.x - p0.x,
            y: p2.y - p0.y,
            z: p2.z - p0.z
        };

        var du1 = t1.x - t0.x;
        var du2 = t2.x - t0.x;
        var dv1 = t1.y - t0.y;
        var dv2 = t2.y - t0.y;

        var det = (du1 * dv2) - (du2 * dv1);

        var tangent = {
            x: (e1.x * dv2 + e2.x * (-dv1)) / det,
            y: (e1.y * dv2 + e2.y * (-dv1)) / det,
            z: (e1.z * dv2 + e2.z * (-dv1)) / det
        };

        var biTangent = {
            x: (e1.x * (-du2) + e2.x * du1) / det,
            y: (e1.y * (-du2) + e2.y * du1) / det,
            z: (e1.z * (-du2) + e2.z * du1) / det
        };

        var normals = {
            x: tangent.y * biTangent.z - tangent.z * biTangent.y,
            y: -(tangent.x * biTangent.z - tangent.z * biTangent.x),
            z: tangent.x * biTangent.y - tangent.y * biTangent.x
        };

        var tangentVector = new Vector3([tangent.x, tangent.y, tangent.z]);
        tangentVector.normalize();

        var biTangentVector = new Vector3([biTangent.x, biTangent.y, biTangent.z]);
        biTangentVector.normalize();

        var normalVector = new Vector3([normals.x, normals.y, normals.z]);
        normalVector.normalize();
```

```
        TBNMatrix[i] = [];
        TBNMatrix[i].push(tangentVector);
        TBNMatrix[i].push(biTangentVector);
        TBNMatrix[i].push(normalVector);
    }
    console.log("TBN Matrix:", TBNMatrix);
}
```

and call it in the DrawScene to see the values being updated.

p0, p1, p2 are points that we get from the vertices. We get t0, t1, t2 from the texture and use them to calculate e1 and e2. From that we can calculate the du1 du2 dv1 dv2 variables to then calculate the values for the matrix.

As we can see, the TBN matrix is being calculated, and the shader is loading both textures, with the normal-map as a bump map and is engraving it on the cube.



This project was run and tested on ***Google Chrome Version 61.0.3163.100 (Official Build) (64-bit)***. Run out of a MAMP stack server. Simply pull the repository and run out of the SRC file.

The SRC folder contains 1 file: **cs411-assignment6-template.html**