BBM414 Introduction to Computer Graphics Lab. Practical # 5- Texture and Lighting

Barış Yıldız
2210356107
Department of Computer Engineering
Hacettepe University
Ankara, Turkey
baris.byildiz@gmail.com

Overview

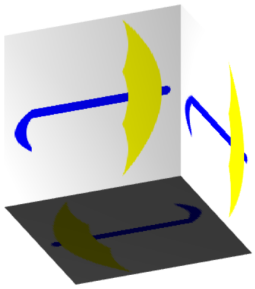
In this experiment, lighting, texture and materials are explored and implemented. The experiment consists of two parts. In the first part a texture is applied to a 3D cube and the lighting defined is adjusted. In the second part, two objects are drawn and are applied separate textures. One of the objects use PBR model while the other uses materials defined in a .mtl file based on Blinn-Phong Ligthing Model.

1 Part 1 - Textured Cube with Lighting

1.1 Applying the Texture

The cube in the given source code only has a gradient color and no texture. The task is to remove this color and instead make the cube take the color of a texture. In this case, we are using the infamous umbrella image we created in prior experiments as texture.

(a) Cube Before Applying Texture



(b) Cube After Applying Texture

Figure 1: Cube in Part 1.

The below Javascript code parts implement this feature. The texture is first loaded using the below procedure. Then, the texture coordinates are populated in the function quad() and kept in an array. Finally, the texture coordinates are given to the vertex shader and then passed to the fragment shader for coloring fragments with the texture.

```
    var umbrellaImage = new Image();
    umbrellaImage.src = "shadedCube_js/umbrella.png";
    umbrellaImage.addEventListener('load', function () {
        var texture = gl.createTexture();
        gl.bindTexture(gl.TEXTURE_2D, texture);
        gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.
            UNSIGNED_BYTE, this);
        gl.generateMipmap(gl.TEXTURE_2D);
        gl.activeTexture(gl.TEXTURE0);
        var textureUniformLoc = gl.getUniformLocation(program,
            "u_texture");
        gl.uniform1i(textureUniformLoc, 0);
    })
```

```
     //Inside function quad(a, b, c, d)

     pointsArray.push(vertices[a]);
     normalsArray.push(normal);
     texCoordsArray.push(vec2(0.0, 0.0));

     pointsArray.push(vertices[b]);
     normalsArray.push(normal);
     texCoordsArray.push(vec2(1.0, 0.0));
     .
     .
     .
```

```
    //Inside function init()

    var texCoordsBuffer = gl.createBuffer();
    gl.bindBuffer(gl.ARRAY_BUFFER, texCoordsBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, flatten(texCoordsArray), gl.
        STATIC_DRAW);

    var vTexCoords = gl.getAttribLocation(program, "vTexCoords
        ");
    gl.vertexAttribPointer(vTexCoords, 2, gl.FLOAT, false, 0,
        0);
    gl.enableVertexAttribArray(vTexCoords);
```

```
//Vertex Shader

#version 300 es
.
.
.
in  vec2 vTexCoords; //texture coordinates are collected from
    JS program.

out vec2 fTexCoords;
.
.
.

void main() {
    .
    .
    .
    fTexCoords = vTexCoords; //passing to the fragment shader.
```

```
}
//Fragment Shader

#version 300 es
precision mediump float;

in vec2 fTexCoords;
.
.
.

uniform sampler2D u_texture; //umbrella texture

void
main()
{
    //applying texture procedure.
    oColor = texture(u_texture, fTexCoords) * fColor;
}
```

1.2 Adjusting Lighting

The light given in the source code is stationary. The task is to allow the user to change the light position using buttons. This is simply done with changing the given *lightPosition* with HTML buttons.

```
    function SetLightPosition(xIncrement, yIncrement,
        zIncrement) {
        lightPosition[0] += xIncrement;
        lightPosition[1] += yIncrement;
        lightPosition[2] += zIncrement;
    }
```

```
    //Use function defined above as button callbacks. (HTML)
    <button onclick="SetLightPosition(0.5,0,0)">Light X+</
        button>
    <button onclick="SetLightPosition(-0.5,0,0)">Light X-</
        button>
    <button onclick="SetLightPosition(0,0.5,0)">Light Y+</
        button>
    <button onclick="SetLightPosition(0,-0.5,0)">Light Y-</
        button>
    <button onclick="SetLightPosition(0,0,0.5)">Light Z+</
        button>
    <button onclick="SetLightPosition(0,0,-0.5)">Light Z-</
        button>
```

2 Part 2 - Scene with Textured Sphere and Plant

The task consists of two main parts: drawing the sphere, drawing the plant and implementing camera controls. The camera controls are taken from Experiment 4 and are abstracted to a Camera class for ease of use, so this is not included in detail in the report.

2.1 Drawing the Sphere

2.1.1 Drawing the Shape

The points of a sphere are given in the formula above (2a). The implementation employs this formula by going over all *theta* and *phi* values in a loop, creating each point.

While pushing points to a point array, the implementation also pushes other vertex attributes such as texture coordinates, normals and tangents. The texture coordinates for each point are defined as the normalized x and y values, the normals are simply the same as points and the formula for the tangents are given below (2b).

$$
\begin{aligned}
x &= x_0 + r \sin\theta \ \cos\varphi \\
y &= y_0 + r \sin\theta \ \sin\varphi \\
z &= z_0 + r \cos\theta
\end{aligned}
$$ [5]

(a) Formula for a point P in the Sphere

$$
\begin{aligned}
T_x &= -r \cos\theta \sin\phi \\
T_y &= 0 \\
T_z &= r \cos\theta \cos\phi
\end{aligned}
$$

(b) Tangent of a point P in the Sphere

Figure 2: Used formulas.

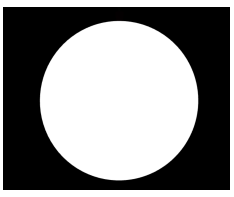
Five different textures are applied to the texture. These are the albedo, normal, ao, metallic and roughness maps given as .png image files. The albedo texture is where the sphere gets its main color. The normal map is where the sphere gets the actual normal vectors at each point. The metallic, ao and roughness maps add extra detail and material properties to the object.

These textures are loaded by following the boilerplate texture loading procedures in WebGL2. After each texture is loaded, they are given as uniform values to the fragment shader of the sphere to be used by the PBR implementation.

2.1.2 PBR

The PBR model is implemented in the fragment shader of the sphere, utilizing several physics formulas such as the **Fresnel-Schlick approximation**, **Trowbridge-Reitz GGX normal distribution function**, **Schlick-GGX geometry function** and **Smith's method**.

The base color used by the model is gathered from the albedo texture. Similarly, the metallic, roughness and ao constants used by the mode are gathered from corresponding texture maps. The normals are gathered from the normal map and also transformed using the *TBN* transformation matrix created with the *normal*, *tangent* and *bitangent* vectors of each fragment. The bitangent vectors are calculated in the vertex shader as the cross product of tangent and normal vectors given as vertex attributes. The *TBN* matrix is also calculated in the vertex shader and passed to the fragment shader.



(a) Sphere Before PBR

(b) Sphere After PBR

Figure 3: Sphere Before and After PBR.

2.2 Lighting and Rotation

The sphere object is finalized by adding the position of the light to its fragment shader and providing the object with *model*, *view* and *projection* matrices for movement. The view and model matrices are gathered from the camera while the model matrix is the multiplication of a **translation matrix** that translates the sphere to the desired world location and a **rotation matrix** that rotates the sphere with respect to its y axis. These matrices are calculated in the *render()* loop of the application.

3 Drawing the Plant Object

3.1 Reading the Files

The plant object is defined by two external files, a .obj file and a .mtl file. Reading of the .obj files are already discussed and implemented in Experiment 4, but the implementation is not directly used in this experiment. This is due to the fact that the .obj file representing the plant object in this experiment has several differences compared to the .obj file of the monkey head from the previous experiment.

One difference is that the file now contains 4 vertices in each *Face* line. This means that each face is defined as a quad consisting of two triangles. One of the triangle uses the first 3 vertices while the other uses all but the second vertex. Another difference is that different materials are used for different groups of faces. These materials are defined in the .mtl file, which means that the .mtl file must be read first. Additionally, in the previous reading implementation of the .obj files, the vertex coordinates were not read properly, this is also fixed now.

The .mtl file contains several materials and inside materials, several coefficients and vectors that represent the **diffuse**, **specular** and **ambient** coefficients in the **Blinn-Phong Lighting Model**. The materials are read and kept as key-value pairs and passed to the function that reads the .obj file. When reading the .obj file afterward, when a new material is being used (usemtl line), the implementation pushes the coefficients of the material to each following face.

3.2 Drawing the Shape and Applying Lighting

Drawing of the shape is the same as in Experiment 4, the only difference is that the face lines now contain four vertex values. This is not an issue if we pass the values to the data array as two triangles. After the .obj file is read correctly, a draw call is issued and the vertex positions are drawn correctly.

Additionally, the implementation applies the provided plant texture to the model while drawing, so not only the positions are drawn, but also the texture is painted on the model using the texture coordinates read from the .obj file.

Each vertex of the plant contains information about the **position**, **texture coordinates**, **normal** (these three properties are read from the .obj file), **ns** (shininess), **ka**, **ks** and **kd** values (these are read from the .mtl file). In order to apply lighting to the model, the implementation uses the normal, ns, ka, ks and kd values and the **Blinn-Phong Lighting Model**. The normal that the model uses doesn't come from the normal that is read from the .obj file, but it is calculated by transforming the normal read from the normal map of the plant with the *TBN* matrix. The TBN matrix is calculated approximately with the normal read from the .obj file and partial derivatives with respect to u and v axes, representing the tangent and bitangent vectors respectively.

(a) Before (b) After

Figure 4: Plant Before and After Texture and Lighting.

References

[1]https://computergraphics.stackexchange.com/questions/5498/compute-sphere-tangent-for-normal-mappingl

[2] https://en.wikipedia.org/wiki/Sphere

[3] https://www.youtube.com/watch?v=hOLLh80hDmw&t=375s

[4] https://www.youtube.com/watch?v=5p0e7YNONr8

[5] https://cs418.cs.illinois.edu/website/text/obj.html

[6]https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model