CS4102 Computer Graphics

Practical 2: Graphics Implementation

220029176

Contents

Part One

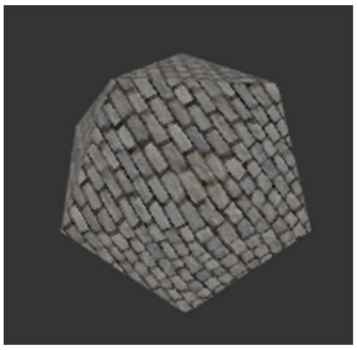


Figure 1: Example for Part One

The shape chosen for this assignment was an icosahedron. It was chosen for its high degree of symmetry, making it easier to manipulate and transform. The vertices and indices were built based on the equations and tables below from Paul Bourke (1993).

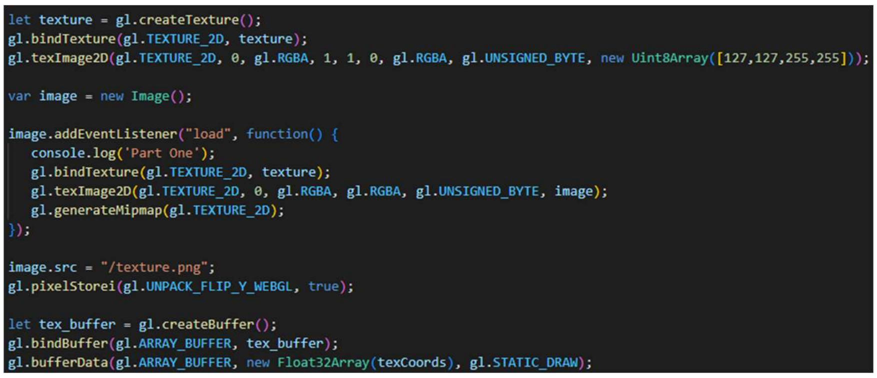
$$\phi = \frac{1+\sqrt{5}}{2}$$

$$a = \phi \;\; b = 1$$

Vertex	**x**	**y**	**z**
V1	0	b	-a
V2	b	a	0
V3	-b	a	0
V4	0	b	a
V5	0	-b	a
V6	-a	0	b
V7	0	-b	-a
V8	a	0	-b
V9	a	0	b
V10	-a	0	-b
V11	b	-a	0
V12	-b	-a	0

Indices		
0	1	2
3	2	1
3	4	5
3	8	4
0	6	7
0	9	6
4	10	11
6	11	10
2	5	9
11	9	5
1	7	8
10	8	7
3	5	2
3	1	8
0	2	9
0	7	1
6	9	11
6	10	7
4	11	5
4	8	10

The trimetric perspective for Part One was obtained by using the 'LookAt' functionality and placing the camera at x=1, y=2 and, z=3. To apply the stone wall texture to the object, a new 2D texture image, an Image object and a texture buffer was created, see Figure 2.



```
let texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 1, 1, 0, gl.RGBA, gl.UNSIGNED_BYTE, new Uint8Array([127,127,255,255]));

var image = new Image();

image.addEventListener("load", function() {
   console.log('Part One');
   gl.bindTexture(gl.TEXTURE_2D, texture);
   gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, image);
   gl.generateMipmap(gl.TEXTURE_2D);
});

image.src = "/texture.png";
gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

let tex_buffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, tex_buffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texCoords), gl.STATIC_DRAW);
```

Figure 2: Texture Processing

The ambient light was generated and applied to all vertices in the vertex shader and corrected with the textures RGB values in the fragment shader, see Figure 1.

Part Two

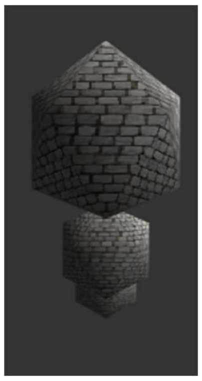
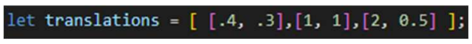


Figure 3: Example for Part Two

For Part Two, three icosahedrons were placed in the scene at different distances from each other as to not overlap.

The animation was obtained by translating the object using the sin() functionality to obtain the desired sinusoidal pattern. To translate the objects at different amplitudes and periods, an array was built to hold two multipliers for each object, see Figure 4. The first multiplier, A, is applied after the sine wave calculation and therefore adjusts the amplitude. The second multiplier, B, multiplies theta before the sine wave is calculated, adjusting the frequency, see Equation 1.



```
let translations = [ [.4, .3],[1, 1],[2, 0.5] ];
```

Figure 4: Amplitudes and Frequencies for Each Object

$$y = Asin(B\theta)$$

Equation 1: Sine Formula

Another requirement of this task was to provide a perspective projection. One-point perspective was implemented using the 'LookAt' functionality where x=6, y=0 and, z=0. This functionality allowed for the one-point perspective projection of the scene, as seen in Figure 3.

To achieve smooth shading with a white point light source, the ambient illumination was combined with a point light source. The ambient light intensity was then decreased to accentuate the directional light while maintaining some ambient light to balance the overall lighting. The point light was generated using the Lambertian reflection model. Firstly, the directional vector, V_D, and the ambient lighting, $I_{Ambient}$, were declared.

$$I_{Ambient} = \begin{bmatrix} 0.1 \\ 0.1 \\ 0.1 \end{bmatrix} \; V_D = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

Then the vertex normal vector was transformed using the normal matrix to maintain the correct orientation relative to the camera and the model.

$$M_{tranformedNormal} = M_{Normal} * V_{Normal}$$

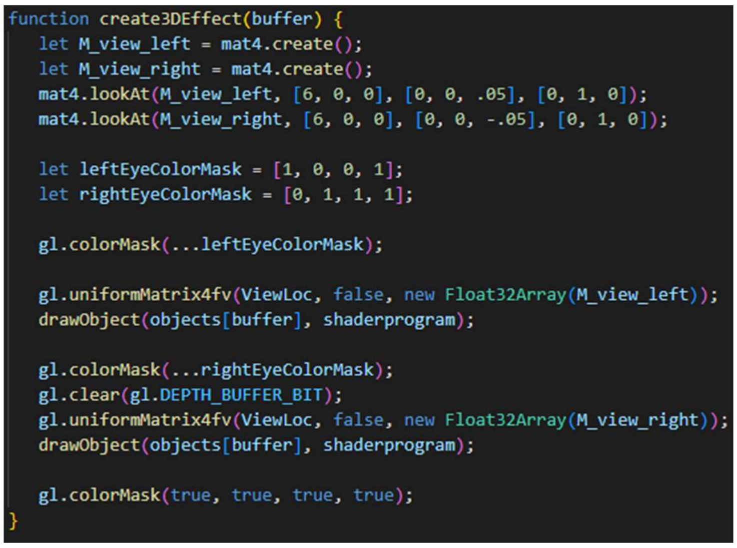
Lambert's Law is then used to calculate the directional lighting by considering the angle between the surface normal and the light direction, see Equation 2. As negative lighting values are prohibited, the dot product of the transformed normal vector and the directional vector is clamped to a minimum value of 0. This is then combined with the ambient light intensity to produce the desired shading on the objects.

$$I_d = I_{Ambient}\left(\max\left((M_{tranformedNormal} \cdot V_D), 0\right)\right)$$

Equation 2: Lambert's Law

Part Three

Stereo render in anaglyph 3D was chosen for the advanced task. After investigating the various resources available it became clear that anaglyphs are generated by combining two images with slightly different perspectives, one for each eye. When viewed through red-cyan glasses, the anaglyph appears to have depth, creating a 3D effect, see Figure 5.



```
function create3DEffect(buffer) {
    let M_view_left = mat4.create();
    let M_view_right = mat4.create();
    mat4.lookAt(M_view_left, [6, 0, 0], [0, 0, .05], [0, 1, 0]);
    mat4.lookAt(M_view_right, [6, 0, 0], [0, 0, -.05], [0, 1, 0]);

    let leftEyeColorMask = [1, 0, 0, 1];
    let rightEyeColorMask = [0, 1, 1, 1];

    gl.colorMask(...leftEyeColorMask);

    gl.uniformMatrix4fv(ViewLoc, false, new Float32Array(M_view_left));
    drawObject(objects[buffer], shaderprogram);

    gl.colorMask(...rightEyeColorMask);
    gl.clear(gl.DEPTH_BUFFER_BIT);
    gl.uniformMatrix4fv(ViewLoc, false, new Float32Array(M_view_right));
    drawObject(objects[buffer], shaderprogram);

    gl.colorMask(true, true, true, true);
}
```

Figure 5: Code Snippet of the 3D Effect Functionality

Firstly, two new perspectives were created with a slightly different perspective generating a parallax effect, which is the foundation of 3D anaglyphs. Then two colour masks were created for the red and blue perspectives. The blue perspective is drawn first by combining the blue colour mask with the left perspective. The depth buffer is then cleared, and the red perspective is drawn. Finally, the colour mask is cleared to not impact the background colour. This functionality is incorporated at the bottom of the main ‘for loop’.

The object positions were adjusted from the one-point perspective seen in Part Two to allow the marker to observe the 3D effect in a more distinct manner.

Part Four

For Part 4, shading using the full Phong model with a normal map was attempted and partially implemented. However, a final complete solution was not obtained due to difficulties with the shading within the fragment shader.

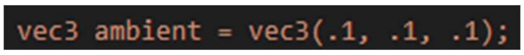
All the lighting code was moved from the vertex shader to the fragment shader to improve the quality of the final render. By moving the lighting calculation, the lighting is calculated per fragment instead of per vertex allowing for more accurate shading.

Phong shading requires the calculation of the ambiences, diffusion and speculation for each fragment. These calculations are then combined to form the final shading model.

Ambiences

$$I_{ia} = L_{ia}R_{ia}$$

Equation 3: Ambient Lighting Calculation

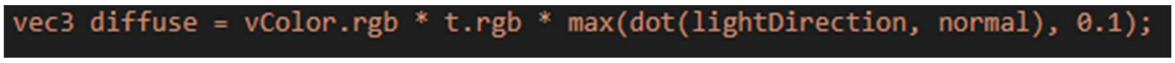


```
vec3 ambient = vec3(.1, .1, .1);
```

Figure 6: Ambient Lighting Calculation in the Code

Ambient lighting represents the constant light in a scene that illuminates all surfaces uniformly, regardless of their orientation. The ambient reflection is traditionally calculated for each of the RGB components and then added up, see Equation 3. However, as we do not know the reflected light intensity for the texture, a vector containing 0.2 for the RGB values will be used, see Figure 6. This is used for the minimum light value for the object.

Diffusion



```
vec3 diffuse = vColor.rgb * t.rgb * max(dot(lightDirection, normal), 0.1);
```

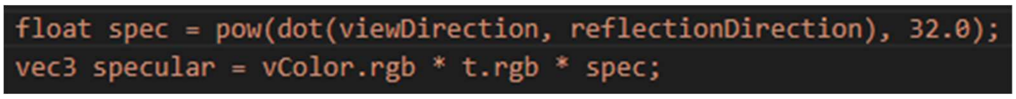
Figure 7: Diffuse Lighting Calculation in the Code

Diffuse lighting represents the light that reflects off the surface in all directions, depending on the angle between the surface normal and the light direction. The diffusion calculation is similar to the previous lighting seen in Parts Two and Three. Therefore, Lambert’s Law will be used to calculate the diffusion terms, see Equation 2. However, this time the normalisation will take place on the normal map instead of the texture map, see Figure 7.

Speculation

$$I_S = k_S L_S (r \cdot v)^a$$

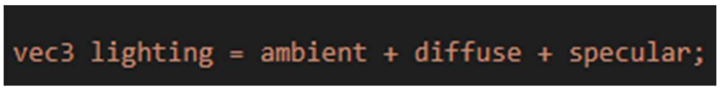
Equation 4: Shininess Coefficient



```
float spec = pow(dot(viewDirection, reflectionDirection), 32.0);
vec3 specular = vColor.rgb * t.rgb * spec;
```

Figure 8: Specular Lighting Calculation in the Code

Specular lighting represents the light that reflects off the surface in a mirror-like manner, producing highlights. The speculation was calculated using the shininess coefficient, see Equation 4 and Figure 8. The final lighting model is formed in Equation 5 which combines the three components, see Figure 9.



```
vec3 lighting = ambient + diffuse + specular;
```

Figure 9: Final Calculation in the Code

$$I = \max\big((M_{tranformedNormal} \cdot V_D), 0\big) + L_{ia}R_{ia} + k_S L_S (r \cdot v)^a$$

Equation 5: Total Illumination Calculation

Additionally, a new normal map buffer was added to the buffer pipeline to allow the normal map to be used. The final object can be seen in Figure 10, with partial Phong shading. If time allowed, further research and implementation would have been completed to achieve a full Phong shading model.

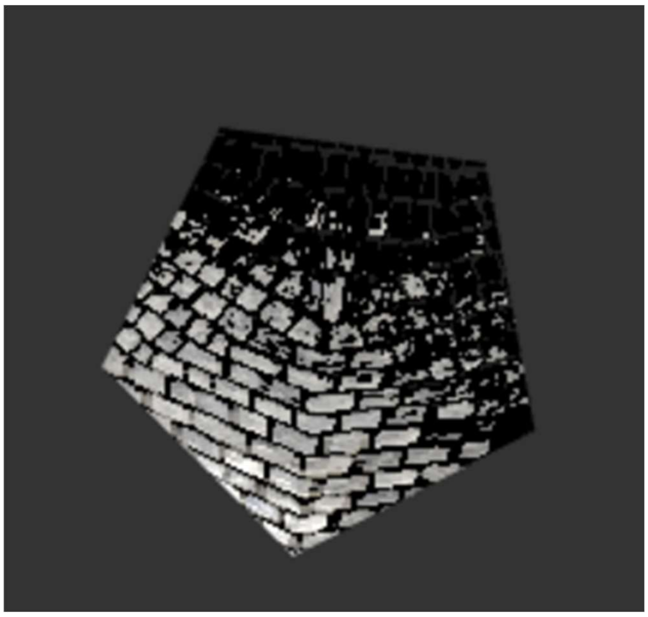


Figure 10: Part Four, a partial implementation of Phong shading

References

Bourke, P. (1993, December 1). *Platonic Solids (Regular polytopes in 3D).* Retrieved from http://paulbourke.net/: http://paulbourke.net/geometry/platonic/