CS405 Project 2 Report

Akif Işıtan

Task 1)

The process of handling odd sized textures involves skipping the generation of mipmaps, since they do not work for textures with odd size, setting texture parameters such as texture wrap to clamp to the edge, and the texture filter being set to linear. The full implementation can be seen in figure 1.

```
// Set texture parameters
if (isPowerOf2(img.width) && isPowerOf2(img.height)) {
  gl.generateMipmap(gl.TEXTURE_2D);
} else {

  /**
   * @Task1 : You should implement this part to accept non power of 2 sized textures
  */

  // If the dimensions are not a power of two, turn off mipmaps and set wrapping to clamp to the edge
  console.log("Non power of 2 sized texture detected. Turning off mipmaps and setting wrapping to clamp to edge.")
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
}
```

Figure 1. Related part of the setTexture function

Task 2)

Fragment shader

In the fragment shader code, it is first checked if the texture is shown and the lighting is enabled. If lighting is not enabled but the texture is visible, the pixel color is set to the color of the texture. If lighting is not enabled and the texture is not visible, the pixel color is set to pure red. If lighting is enabled and the texture is visible, first the texture color is obtained and the surface normal is calculated. Next, the diffuse light is calculated by finding the dot product between the normalized surface normal and the direction from the fragment to the light. The result is multiplied by the color of the vertex to get the diffuse light on the vertex. Then the ambient light intensity is applied and added up with the diffuse light to obtain the final color, which is then applied as lighting to the object. The full implementation is shown in figure 2.

```
// Fragment shader source code
/**
 * @Task2 : You should update the fragment shader to handle the lighting
 */
const meshFS = `
      precision mediump float;

      uniform bool showTex;
      uniform bool enableLighting;
      uniform sampler2D tex;
      uniform vec3 color;
      uniform vec3 lightPos;
      uniform float ambient;

      varying vec2 v_texCoord;
      varying vec3 v_normal;

      void main()
      {
        if(showTex && enableLighting) {
          vec4 texColor = texture2D(tex, v_texCoord); // Get the color from the texture
          vec3 norm = normalize(v_normal); // Normalize the surface normal

          // Calculate the diffuse light
          vec3 lightDir = normalize(lightPos); // Direction from the fragment to the light
          float diff = max(dot(norm, lightDir), 0.0); // Calculate the diffuse impact by taking the dot product of the vector to the light and the normal
          vec3 diffuse = diff * texColor.rgb; // Multiply the impact by the color of the vertex to get the diffuse light on the vertex

          // Calculate the ambient light
          vec3 ambientLight = vec3(ambient, ambient, ambient); // Ambient light is a uniform value applied across all surfaces

          // Combine the ambient and diffuse lighting and apply it to the texture color
          vec3 finalColor = ambientLight + diffuse; // Combine the ambient and diffuse lighting
          gl_FragColor = vec4(finalColor, 1.0) * texColor; // Apply the lighting
        }
        else if(showTex) {
          gl_FragColor = texture2D(tex, v_texCoord);
        }
        else {
          gl_FragColor =  vec4(1.0, 0, 0, 1.0);
        }
      }`;
```

Figure 2. Fragment shader implementation

Constructor

In the constructor method, the shader uniforms and attributes are initialized by referencing them via the `gl` context and their name in the shader program, variables for ambient light intensity are set, the normal buffer is created and a variable is declared to track lighting enabled / disabled state. The full implementation can be seen in figure 3.

```
/**
 * @Task2 : You should initialize the required variables for lighting here
 */

this.lightPosLoc = gl.getUniformLocation(this.prog, 'lightPos'); // Location of light position uniform
this.ambientLoc = gl.getUniformLocation(this.prog, 'ambient'); // Location of ambient light uniform
this.enableLightingLoc = gl.getUniformLocation(this.prog, 'enableLighting'); // Location of the lighting enable/disable uniform
this.normalLoc = gl.getAttribLocation(this.prog, 'normal'); // Location of the vertex normal attribute

this.ambient = 0.5; // Ambient light intensity
this.normalBuffer = gl.createBuffer(); // Buffer for vertex normals
this.lightingEnabled = false; // Track the state of lighting
```

Figure 3. Related part of the constructor

setMesh

The required steps for enabling lighting as far as this function is concerned are binding the buffers and setting the buffer data using the provided vertex position, texture coordinates and normal coordinates. The full function implementation can be seen in figure 4.

```
setMesh(vertPos, texCoords, normalCoords) {
  gl.bindBuffer(gl.ARRAY_BUFFER, this.vertbuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertPos), gl.STATIC_DRAW);

  // update texture coordinates
  gl.bindBuffer(gl.ARRAY_BUFFER, this.texbuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(texCoords), gl.STATIC_DRAW);

  this.numTriangles = vertPos.length / 3;

  /**
   * @Task2 : You should update the rest of this function to handle the lighting
   */

  // Bind and set the normal buffer data
  if (normalCoords) {
    gl.bindBuffer(gl.ARRAY_BUFFER, this.normalBuffer);
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(normalCoords), gl.STATIC_DRAW);
  }

}
```

Figure 4. setMesh function implementation

Draw

Lighting is handled in the draw method by first checking if lighting is enabled. If lighting is enabled via the checkbox, first an arbitrary Z value for the light is set, chosen as -5.0 for this context such that the light is in front of the object. The light's X and Y coordinates are initialized as 1 and 1 in a separate context and modified by the arrow keys, which are then used in this function. First the lighting flag is enabled in the shader to prevent synchronization issues, then the light position uniform shader variable is set to the X, Y and Z values of the light position. The ambient light intensity is set, and finally the buffer is bound to the normal data. The relevant snippet is shown in figure 5.

```
/**
 * @Task2 : You should update this function to handle the lighting
 */

if (this.lightingEnabled) {
  const lightZ = -5.0;
  console.log(`Light position: (${lightX}, ${lightY}, ${lightZ})`);
  // Enable the lighting flag in the shader
  gl.uniform1i(this.enableLightingLoc, true);
  // Set the light position uniform
  gl.uniform3f(this.lightPosLoc, lightX, lightY, lightZ);
  // Set the ambient light uniform
  gl.uniform1f(this.ambientLoc, this.ambient);
  // Bind and point to the normal data
  gl.bindBuffer(gl.ARRAY_BUFFER, this.normalBuffer);
  gl.enableVertexAttribArray(this.normalLoc);
  gl.vertexAttribPointer(this.normalLoc, 3, gl.FLOAT, false, 0, 0);
}
```

Figure 5. Draw method lighting snippet

setAmbientLight

This function is used to increase and decrease the ambient light intensity via a slider in the page. It takes the value from the slider and uses the `gl` context to get access to the shader program to modify the uniform shader variable bound to this.ambientLoc. Full implementation for reference below in figure 6.

```
setAmbientLight(ambient) {
  /**
   * @Task2 : You should implement the lighting and implement this function
   */

  console.log("Changed ambient light intensity:", ambient)
  // Set the uniform to the provided ambient light intensity
  this.ambient = ambient;
  gl.useProgram(this.prog);
  gl.uniform1f(this.ambientLoc, ambient);
}
```

Figure 6. setAmbientLight function implementation

enableLighting

This function is used to enable / disable lighting depending on the `show` variable via a checkbox in the page. It takes the value from the checkbox and uses the program as reference to modify the uniform shader variable bound to this.enableLightingLoc. Full implementation of this function can be seen in figure 7.

```
enableLighting(show) {
    /**
     * @Task2 : You should implement the lighting and implement this function
     */

    console.log("Changed lighting state: ", show ? "enabled" : "disabled");
    // Set the uniform to enable or disable lighting in the shader
    this.lightingEnabled = show;
    gl.useProgram(this.prog);
    gl.uniform1i(this.enableLightingLoc, show ? 1 : 0);
}
```

Figure 7. enableLighting function implementation

The implementation of these methods allows for lighting to be disabled, enabled, increased and moved. The results can be seen on the same object shown in figures 8, 9, 10 and 11 with different parameters modified.

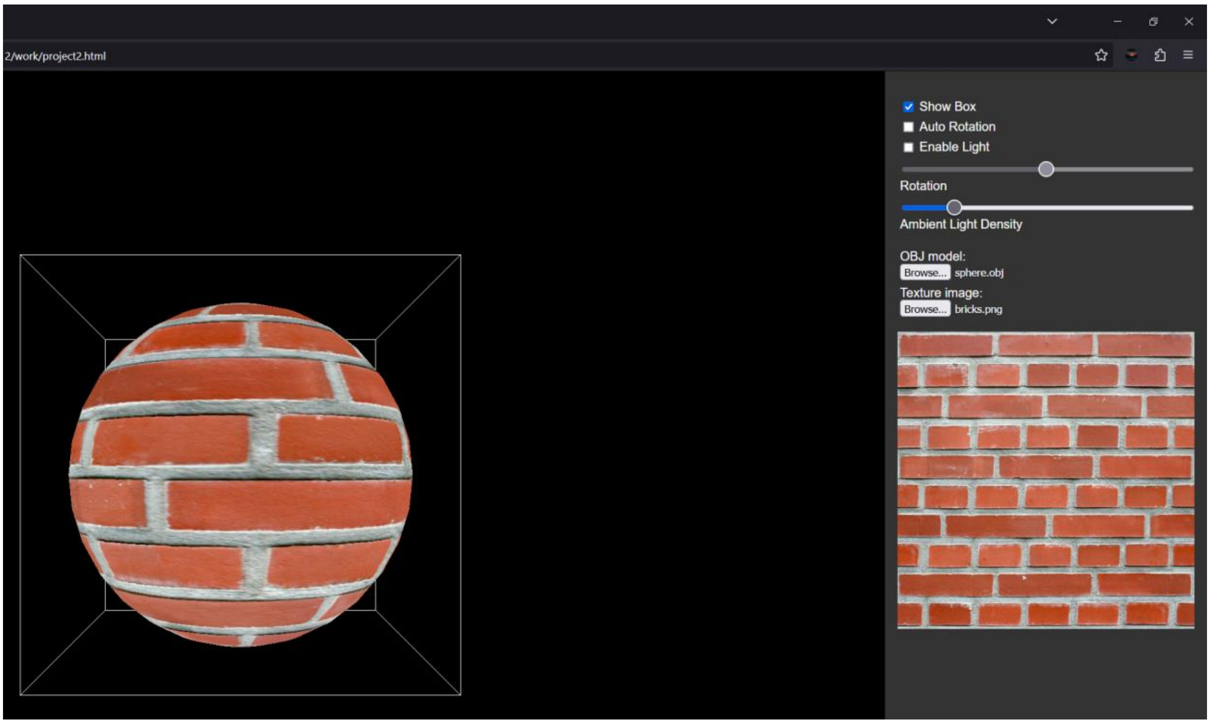

Figure 8. Object displayed with lighting disabled

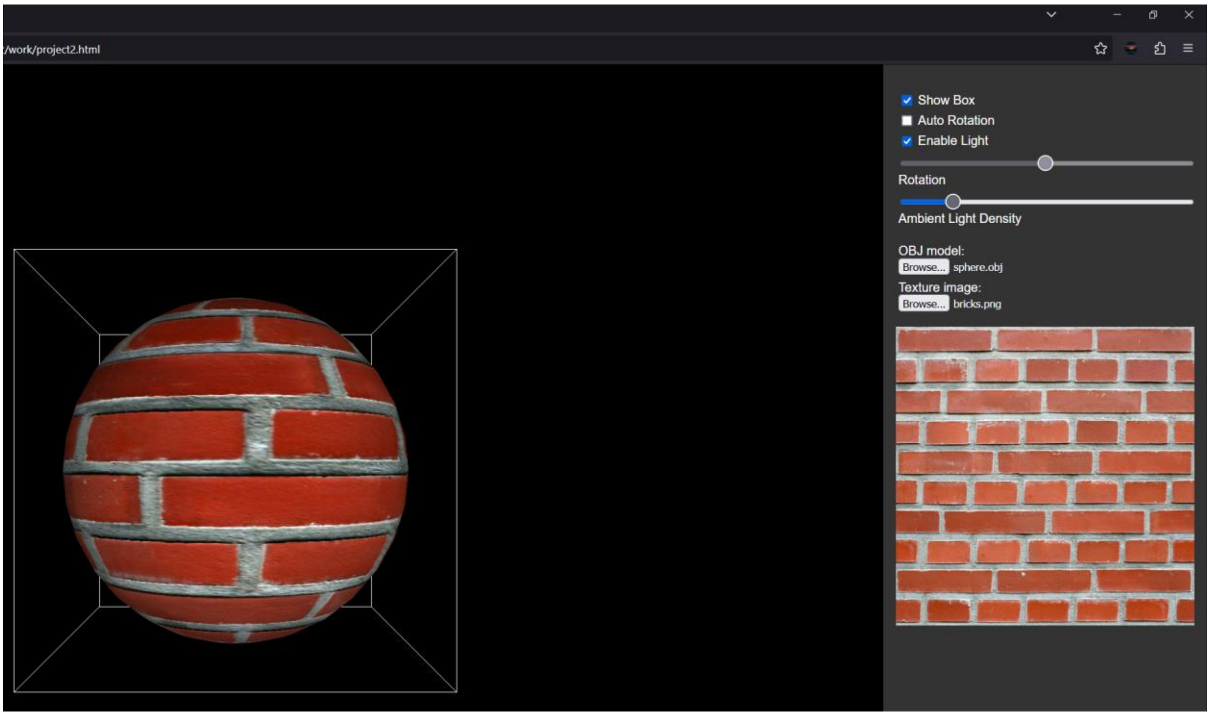

Figure 9. Object displayed with lighting enabled, low ambient light density

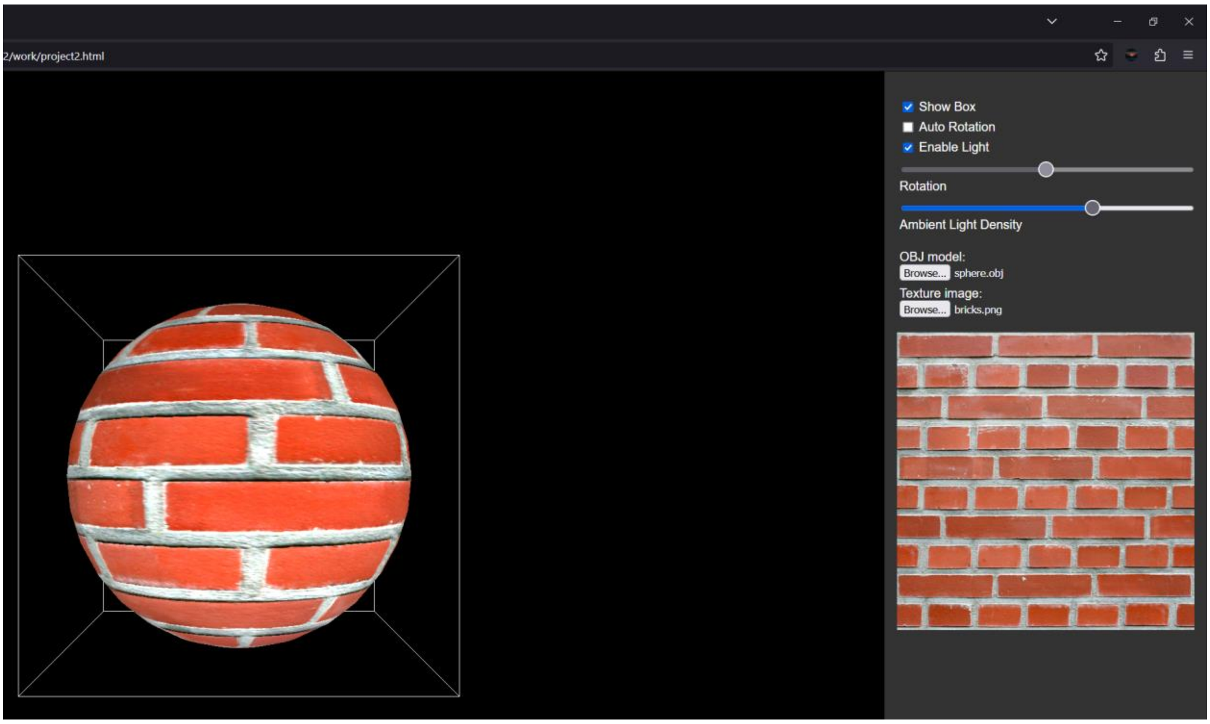

Figure 10. Object displayed with lighting enabled, high ambient light density

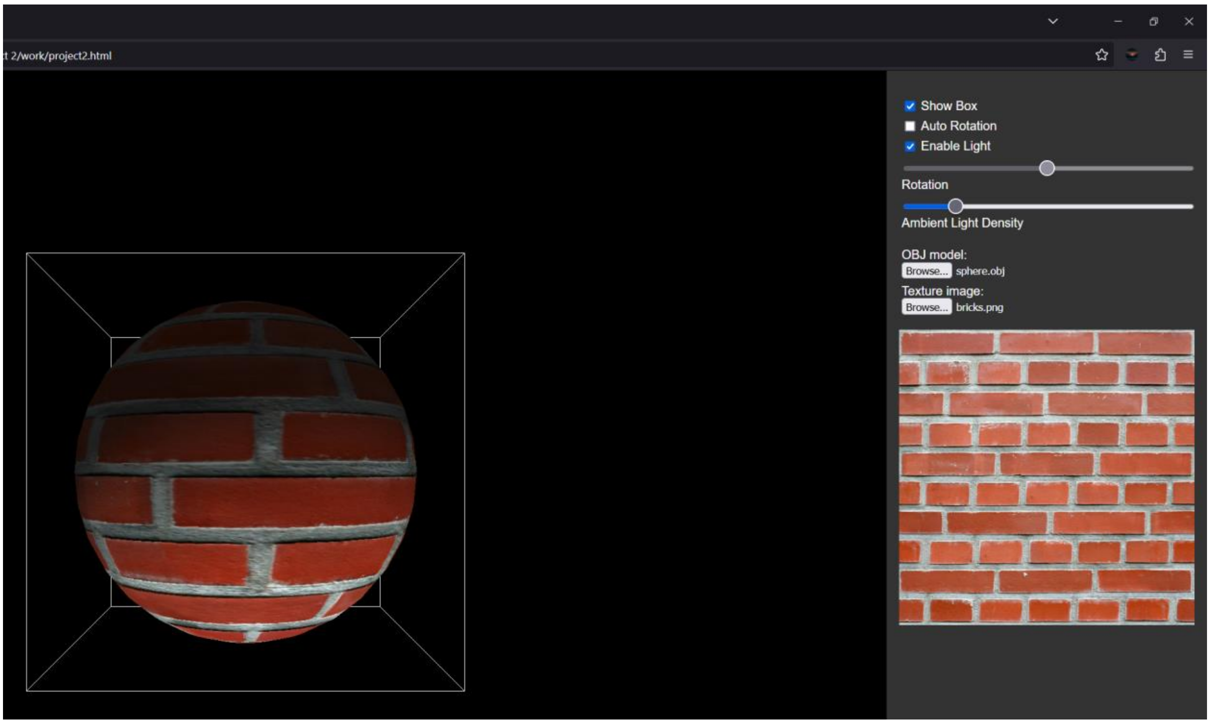

Figure 11. Object displayed with lighting enabled, light position moved towards the bottom