Computer Graphics Project

Razvan Popan

January 2025

Contents

1 Introduction

The following project is made with the purpose of developing skills in programming in GLSL, C++, developing problem solving skills, learn something new and have fun :). This project contains object imported in Blender then exported as .obj as to use them in the OpenGL application. In the OpenGL application we will use shaders to perform different kinds of computations to improve the photorealism of the scene, ranging from calculating light(directional light, point light and spot light) to calculating shadows, calculating fog and using gamma correction.

2 Scenario

The scene takes inspiration from the Silent Hill game(I tried), you are put on a empty road inside a foggy forest, and your goal is to do nothing. Enjoy the moment, the silence in a place where no one can hear you, bother you, you are alone with your thoughts, and a spot light, and a directional light, and some shadows.

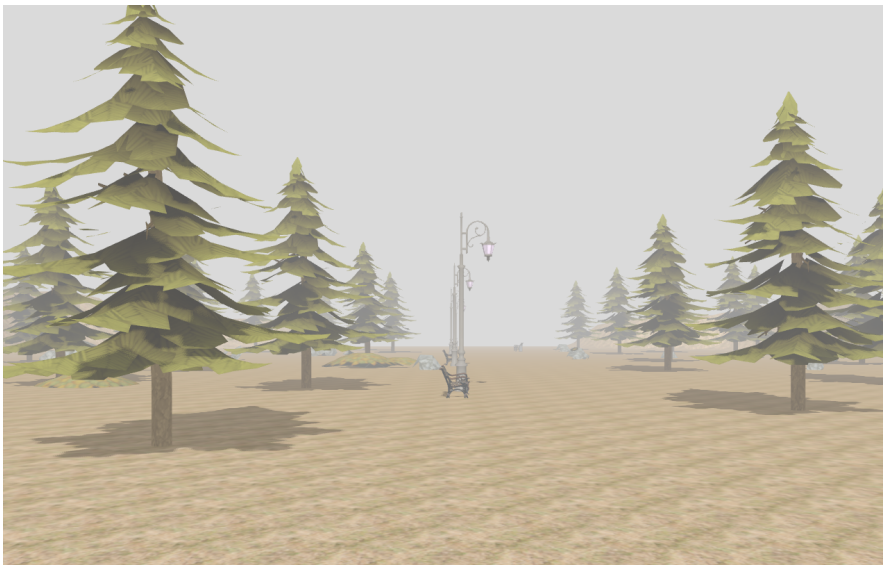

Figure 1: Scene

3 Implementation

3.1 Main

```
int main(int argc, const char * argv[]) {

    try {
        initOpenGLWindow();
    } catch (const std::exception& e) {
        std::cerr << e.what() << std::endl;
        return EXIT_FAILURE;
    }

    initOpenGLState();
    initModels();
    initShaders();
    initShadowFBO();
    initUniforms();
    initUniformsPointLight();
    initUniformSpotLight();
    setWindowCallbacks();

        glCheckError();
        // application loop
        while (!glfwWindowShouldClose(myWindow.getWindow()))
             {

        processMovement();

        renderScene();
                glfwPollEvents();

                glfwSwapBuffers(myWindow.getWindow());

                glCheckError();
        }

        cleanup();

    return EXIT_SUCCESS;
}
```

This is the main function of the application, first we initialize the window, then the stat of OpenGL(enable Back-face Culling etc.), initialize models, initialize shader, initialize Shadow Frame Buffer for calculating shadows using the z-depth buffer, initializing uniforms for the 3 lights(directional, spotlight, positional). What isof interest here is the loop of the application, and in it we will focus on mainly the movement processing and scene rendering.

3.2 Movement Processing

For this part we will use 3 functions, one for processing movement, one for processing keyboard events and one for mouse events.

```
    void mouseCallback(GLFWwindow* window, double xpos,
        double ypos)
{
    if (!first_mouse) {
        x_pos_cursor = xpos;
        y_pos_cursor = ypos;
        first_mouse = true;
    }

    yaw = (-xpos + x_pos_cursor) * sensitivity;
    pitch = (-ypos + y_pos_cursor) * sensitivity;

    myCamera.rotate(pitch, yaw);

    myBasicShader.useShaderProgram();
    glUniformMatrix4fv(glGetUniformLocation(myBasicShader.
        shaderProgram, "view"), 1, GL_FALSE, glm::value_ptr(
        myCamera.getViewMatrix()));
    glUniform3fv(glGetUniformLocation(myBasicShader.
        shaderProgram, "cameraPositon"), 1, glm::value_ptr(
        myCamera.getPosition()));
    glUniform3fv(glGetUniformLocation(myBasicShader.
        shaderProgram, "spotLightPos"), 1, glm::value_ptr(
        myCamera.getPosition()));
    glUniform3fv(glGetUniformLocation(myBasicShader.
        shaderProgram, "spotLightDir"), 1, glm::value_ptr(
        myCamera.getDirection()));

    x_pos_cursor = xpos;
    y_pos_cursor = ypos;
}
```

This function deals with mouse events, it is used to move the camera, to face it in other directions. At each event we calculate the offset between x and y coordinates of two successive events, multiply it with a chosen sensitivity, and then rotate the camera based on this offsets. After each step we send the new view matrix to the shader, and the positions of the camera for the spotlight(more on this later).

```
    void keyboardCallback(GLFWwindow* window, int key, int
        scancode, int action, int mode) {
        if (key == GLFW_KEY_ESCAPE && action == GLFW_PRESS)
            {
        glfwSetWindowShouldClose(window, GL_TRUE);
    }
        if (key >= 0 && key < 1024) {
        if (action == GLFW_PRESS) {
            pressedKeys[key] = true;
        } else if (action == GLFW_RELEASE) {
            pressedKeys[key] = false;
        }
    }
    if (key == GLFW_KEY_N && action == GLFW_PRESS) {
        start_animation1 = true;
    }
    if (key == GLFW_KEY_T && action == GLFW_PRESS) {
        switch (wireframeView)
        {
        case 0:
            glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
            break;
        case 1:
            glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
            break;
        case 2:
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
            wireframeView = -1;
            break;
        default:
            break;
        }
        wireframeView += 1;
    }
    if (key == GLFW_KEY_M && action == GLFW_PRESS) {
        showSceneFromLight = !showSceneFromLight;
    }
    if (key == GLFW_KEY_V && action == GLFW_PRESS) {
        ifSpotLight ^= 1;
        myBasicShader.useShaderProgram();
        glUniform1i(glGetUniformLocation(myBasicShader.
            shaderProgram, "ifSpotLight"), ifSpotLight);
    }
}
```

This function handles input form the keyboard. It is used to start the animation for presenting the scene and, close the app or select between different view modes of the scene.

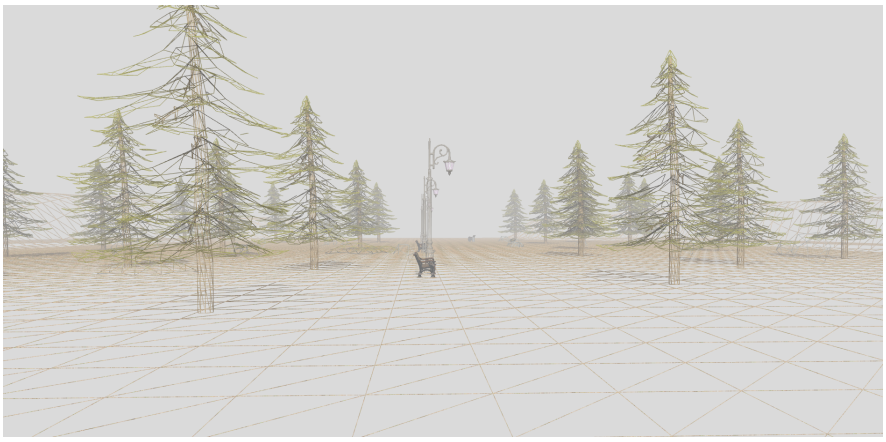

Figure 2: Wireframe

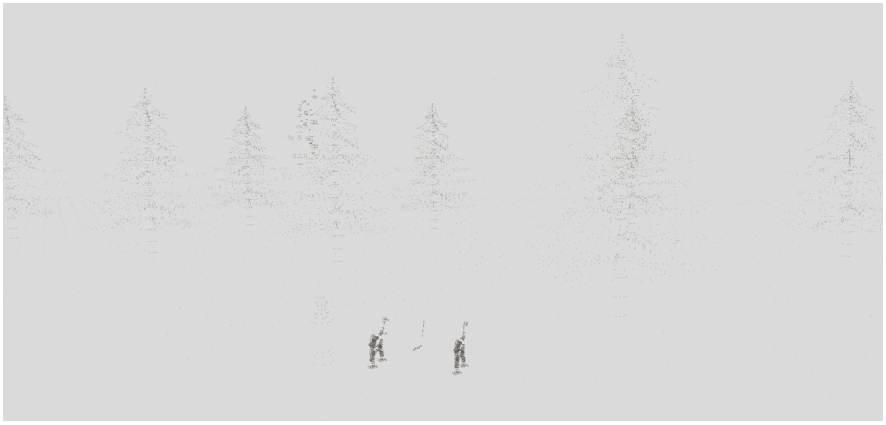

Figure 3: Points

```
    void processMovement() {
        if (pressedKeys[GLFW_KEY_W]) {
          start_animation1 = false;
       start_animation2 = false;
       start_animation3 = false;
         start_animation4 = false;
         anim_cnt = 0.0f;
                myCamera.move(gps::MOVE_FORWARD, cameraSpeed
                    );

         view = myCamera.getViewMatrix();
         myBasicShader.useShaderProgram();
         glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
```

```
            value_ptr(view));
13          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
14          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
15          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
16          normalMatrix = glm::mat3(glm::inverseTranspose(view*
                model));
17          }
18
19          if (pressedKeys[GLFW_KEY_S]) {
20          start_animation1 = false;
21          start_animation2 = false;
22          start_animation3 = false;
23          start_animation4 = false;
24          anim_cnt = 0.0f;
25                  myCamera.move(gps::MOVE_BACKWARD,
                        cameraSpeed);
26
27          view = myCamera.getViewMatrix();
28          myBasicShader.useShaderProgram();
29          glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
                value_ptr(view));
30          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
31          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
32          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
33
34          normalMatrix = glm::mat3(glm::inverseTranspose(view*
                model));
35          }
36
37          if (pressedKeys[GLFW_KEY_A]) {
38          start_animation1 = false;
39          start_animation2 = false;
40          start_animation3 = false;
41          start_animation4 = false;
42          anim_cnt = 0.0f;
43          myCamera.move(gps::MOVE_LEFT, cameraSpeed);
44
45          view = myCamera.getViewMatrix();
```

```
46          myBasicShader.useShaderProgram();
47          glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
                value_ptr(view));
48          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
49          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
50          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
51          normalMatrix = glm::mat3(glm::inverseTranspose(view*
                model));
52          }
53
54          if (pressedKeys[GLFW_KEY_D]) {
55          start_animation1 = false;
56          start_animation2 = false;
57          start_animation3 = false;
58          start_animation4 = false;
59          anim_cnt = 0.0f;
60                  myCamera.move(gps::MOVE_RIGHT, cameraSpeed);
61          view = myCamera.getViewMatrix();
62          myBasicShader.useShaderProgram();
63          glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
                value_ptr(view));
64          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
65          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
66          glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
67          normalMatrix = glm::mat3(glm::inverseTranspose(view*
                model));
68          }
69
70      if (pressedKeys[GLFW_KEY_Q]) {
71          start_animation1 = false;
72          start_animation2 = false;
73          start_animation3 = false;
74          start_animation4 = false;
75          anim_cnt = 0.0f;
76          angle -= 1.0f;
77          model = glm::rotate(glm::mat4(1.0f), glm::radians(
                angle), glm::vec3(0, 1, 0));
78
```

```
79          normalMatrix = glm::mat3(glm::inverseTranspose(view*
                model));
80      }
81
82      if (pressedKeys[GLFW_KEY_E]) {
83          start_animation1 = false;
84          start_animation2 = false;
85          start_animation3 = false;
86          start_animation4 = false;
87          anim_cnt = 0.0f;
88          angle += 1.0f;
89          model = glm::rotate(glm::mat4(1.0f), glm::radians(
                angle), glm::vec3(0, 1, 0));
90
91          normalMatrix = glm::mat3(glm::inverseTranspose(view*
                model));
92      }
93
94      if (start_animation1) {
95          myCamera.move(gps::MOVE_FORWARD, cameraSpeed/2);
96
97          view = myCamera.getViewMatrix();
98          myBasicShader.useShaderProgram();
99          glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
                value_ptr(view));
100         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
101         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
102         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
103
104         normalMatrix = glm::mat3(glm::inverseTranspose(view
                * model));
105         anim_cnt += 0.01f;
106         if (anim_cnt > 2.4f) {
107             anim_cnt = 0.0f;
108             start_animation1 = false;
109             start_animation2 = true;
110         }
111     }
112
113     if (start_animation2) {
114         anim_angle = 0.3f;
115         myCamera.rotate(0.0f , anim_angle);
116
117         view = myCamera.getViewMatrix();
```

```
118         myBasicShader.useShaderProgram();
119         glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
                value_ptr(view));
120         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
121         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
122         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
123
124         normalMatrix = glm::mat3(glm::inverseTranspose(view
                * model));
125         anim_cnt += 0.01f;
126         if (anim_cnt > 7.4f) {
127             anim_cnt = 0.0f;
128             start_animation2 = false;
129             start_animation3 = true;
130         }
131     }
132
133     if (start_animation3) {
134         anim_angle = 0.3f;
135         if (anim_cnt < 2.5f) {
136             myCamera.rotate(-anim_angle, 0.0f);
137         }
138         myCamera.move(gps::MOVE_BACKWARD, cameraSpeed / 9);
139
140         view = myCamera.getViewMatrix();
141         myBasicShader.useShaderProgram();
142         glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
                value_ptr(view));
143         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "cameraPositon"), 1, glm::
                value_ptr(myCamera.getPosition()));
144         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightPos"), 1, glm::value_ptr
                (myCamera.getPosition()));
145         glUniform3fv(glGetUniformLocation(myBasicShader.
                shaderProgram, "spotLightDir"), 1, glm::value_ptr
                (myCamera.getDirection()));
146
147         normalMatrix = glm::mat3(glm::inverseTranspose(view
                * model));
148
149         anim_cnt += 0.01f;
150         if (anim_cnt > 14.4f) {
151             anim_cnt = 0.0f;
```

```
            start_animation3 = false;
            start_animation4 = true;
        }
    }

    if (start_animation4) {
        anim_angle = 0.3f;
        if (anim_cnt > 12.0f) {
            myCamera.rotate(anim_angle, 0.0f);
        }
        myCamera.move(gps::MOVE_FORWARD, cameraSpeed / 9);

        view = myCamera.getViewMatrix();
        myBasicShader.useShaderProgram();
        glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::
            value_ptr(view));
        glUniform3fv(glGetUniformLocation(myBasicShader.
            shaderProgram, "cameraPositon"), 1, glm::
            value_ptr(myCamera.getPosition()));
        glUniform3fv(glGetUniformLocation(myBasicShader.
            shaderProgram, "spotLightPos"), 1, glm::value_ptr
            (myCamera.getPosition()));
        glUniform3fv(glGetUniformLocation(myBasicShader.
            shaderProgram, "spotLightDir"), 1, glm::value_ptr
            (myCamera.getDirection()));

        normalMatrix = glm::mat3(glm::inverseTranspose(view
            * model));

        anim_cnt += 0.01f;
        if (anim_cnt > 14.4f) {
            anim_cnt = 0.0f;
            start_animation4 = false;
        }
    }

    glCheckError();
    if (pressedKeys[GLFW_KEY_B]) {
        rotateLight();
    }
    glCheckError();
}
```

This function is called every time in the loop of the application, to handle different movements, for the animtion and for the camera, which can move forward, backwards, left, right, look in all directions, rotate the scene and rotate the light.

```
void rotateLight() {
    glm::mat4 mat(1.0f);
    mat = glm::rotate(mat, 0.01f, glm::vec3(1.0f, 0.0f, 0.0f
        ));
    lightDir = glm::vec3(mat * glm::vec4(lightDir,1.0f));
    myBasicShader.useShaderProgram();
    lightDirLoc = glGetUniformLocation(myBasicShader.
        shaderProgram, "lightDir");
    // send light dir to shader
    glUniform3fv(lightDirLoc, 1, glm::value_ptr(lightDir));
    glCheckError();
}
```

Implementation of the camera class:

```
    #include "Camera.hpp"

namespace gps {

    //Camera constructor
    Camera::Camera(glm::vec3 cameraPosition, glm::vec3
        cameraTarget, glm::vec3 cameraUp) {
        this->cameraPosition = cameraPosition;
        this->cameraTarget = cameraTarget;
        this->cameraUpDirection = glm::normalize(cameraUp);
        this->cameraFrontDirection = glm::normalize(
            cameraTarget - cameraPosition);
        this->cameraRightDirection = glm::normalize(glm::
            cross(this->cameraFrontDirection, this->
            cameraUpDirection));
    }

    //return the view matrix, using the glm::lookAt()
        function
    glm::mat4 Camera::getViewMatrix() {
        return glm::mat4(glm::lookAt(this->cameraPosition,
            this->cameraTarget, this->cameraUpDirection));
    }

    //update the camera internal parameters following a
        camera move event
    void Camera::move(MOVE_DIRECTION direction, float speed)
         {
        switch (direction)
        {

        case gps::MOVE_FORWARD:
            this->cameraPosition += speed * this->
                cameraFrontDirection;
            break;
```

```
27          case gps::MOVE_BACKWARD:
28              this->cameraPosition -= speed * this->
                    cameraFrontDirection;
29              break;
30          case gps::MOVE_RIGHT:
31              this->cameraPosition += speed * this->
                    cameraRightDirection;
32              break;
33          case gps::MOVE_LEFT:
34              this->cameraPosition -= speed * this->
                    cameraRightDirection;
35              break;
36          default:
37              break;
38          }
39
40          this->cameraTarget = this->cameraPosition + this->
                cameraFrontDirection;
41      }
42
43      //update the camera internal parameters following a
            camera rotate event
44      //yaw - camera rotation around the y axis
45      //pitch - camera rotation around the x axis
46      void Camera::rotate(float pitch, float yaw) {
47
48          glm::mat4 transformation_camera = glm::mat4(1.0f);
49          transformation_camera = glm::translate(
                transformation_camera, -this->cameraPosition);
50          transformation_camera = glm::rotate(
                transformation_camera, glm::radians(pitch), this
                ->cameraRightDirection);
51          transformation_camera = glm::rotate(
                transformation_camera, glm::radians(yaw), this->
                cameraUpDirection);
52          transformation_camera = glm::translate(
                transformation_camera, this->cameraPosition);
53
54          //this->cameraUpDirection = glm::normalize(glm::vec3
                (transformation_camera * glm::vec4(this->
                cameraUpDirection, 0)));
55          this->cameraFrontDirection = glm::normalize(glm::
                vec3(transformation_camera * glm::vec4(this->
                cameraFrontDirection, 0)));
56          this->cameraRightDirection = glm::normalize(glm::
                cross(this->cameraFrontDirection, this->
                cameraUpDirection));
57
58          this->cameraTarget = this->cameraPosition + this->
                cameraFrontDirection;
```

```
59
60          //TODO
61      }
62
63      glm::vec3 Camera::getPosition() {
64          return this->cameraPosition;
65      }
66
67      glm::vec3 Camera::getDirection(){
68          return this->cameraFrontDirection;
69      }
70  }
```

3.3 Rendering

```
1   void renderScene() {
2       glCheckError();
3       shadowPass();
4       if (showSceneFromLight) {
5           showDepthMap();
6       }
7       else {
8               lightPass();
9        }
10  }
```

The rendering is split into two phases, first is the shadow pass, where we compute the depth map and then in the light pass stage, we compare the z values of the fragments with the z from the depth buffer, and based on the comparison we can say that the fragment is in the shadow of the object or not. For the shadow map technique to work we first have to initialize a Frame Buffer Object, and bind the texture to the created depth map.

```
1       void initShadowFBO() {
2
3       glGenFramebuffers(1, &shadowMapFBO); //generate shadow
            map buffer
4       glGenTextures(1, &depthMapTexture); //generate texture
            for the shadow map
5       glBindTexture(GL_TEXTURE_2D, depthMapTexture);
6
7       glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
            SHADOW_WIDTH, SHADOW_HEIGHT, 0, GL_DEPTH_COMPONENT,
            GL_FLOAT, NULL);
8       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
            GL_NEAREST);
9       glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
            GL_NEAREST);
```

```
10
11      float borderColor[4] = { 1.0f,1.0f,1.0f,1.0f };
12      glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR,
            borderColor);
13      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
            GL_CLAMP_TO_BORDER);
14      glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
            GL_CLAMP_TO_BORDER);
15
16
17      glBindFramebuffer(GL_FRAMEBUFFER, shadowMapFBO);
18      glFramebufferTexture2D(GL_FRAMEBUFFER,
            GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, depthMapTexture,
            0);
19
20
21      glDrawBuffer(GL_NONE);
22      glReadBuffer(GL_NONE);
23
24      glBindFramebuffer(GL_FRAMEBUFFER, 0);
25
26  }
```

```
1       void shadowPass() {
2
3       depthMapShader.useShaderProgram();
4       glUniformMatrix4fv(glGetUniformLocation(depthMapShader.
            shaderProgram, "lightSpaceTrMatrix"),1,GL_FALSE,glm::
            value_ptr(computeLightSpaceTrMatrix()));
5       glViewport(0, 0, SHADOW_WIDTH, SHADOW_HEIGHT);
6       glBindFramebuffer(GL_FRAMEBUFFER , shadowMapFBO);
7       glClear(GL_DEPTH_BUFFER_BIT);
8       glCullFace(GL_BACK);
9       drawObjects(depthMapShader, true);
10      glCullFace(GL_FRONT);
11      drawObjects(depthMapShader, true);
12      //render back and front for solving Peter Panning
13      glBindFramebuffer(GL_FRAMEBUFFER, 0);
14      glCullFace(GL_BACK);// set back to culling the backface
15
16
17  }
```

Objects are drawn twice using backface culling and front face culling to help prevent the Peter Panning artifact (shadows are moved). In the fragment shader we will use a Percentage Closer Filtering that takes into account the neighbouring fragments when looking at the depth map, and based on that we compute a shadow factor, to have smooth edge shadows.

```
    float computeShadow(){

        vec3 normalizedCoords = fragPosLightSpace.xyz /
            fragPosLightSpace.w;

        normalizedCoords = normalizedCoords * 0.5 + 0.5;

    vec2 TexelSize = vec2(1.0f/2048.0f,1.0f/2048.0f);

        if (normalizedCoords.z > 1.0f)
                return 0.0f;


    float bias = max(0.05f * (1.0f - dot(normalizedCoords,
        lightDir)), 0.005f);
    float shadow = 0.0f;

    float currentDepth = normalizedCoords.z;
    for(int v=-1;v<=1;v++)
        for(int u=-1;u<=1;u++){
            vec2 offset = vec2(u,v) * TexelSize;
            float closestDepth = texture(shadowMap,
                normalizedCoords.xy + offset).r;

            if(currentDepth - bias > closestDepth){
                shadow += 1.0f;
            }
            else{
                shadow += 0.0f;
            }

        }

        return (shadow/9.0f);
}
```

In the shadow pass stage, we have "moved" the camera into the "position" of the directional light, and applied an ortographic projection.In the light pass stage, we will compute the intensity of each fragment as the sum of intensities produced by each light source. The intensity is split into three components, ambient, diffuse and specular.

```
void lightPass() {

    glViewport(0, 0, myWindow.getWindowDimensions().width,
        myWindow.getWindowDimensions().height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    myBasicShader.useShaderProgram();

```

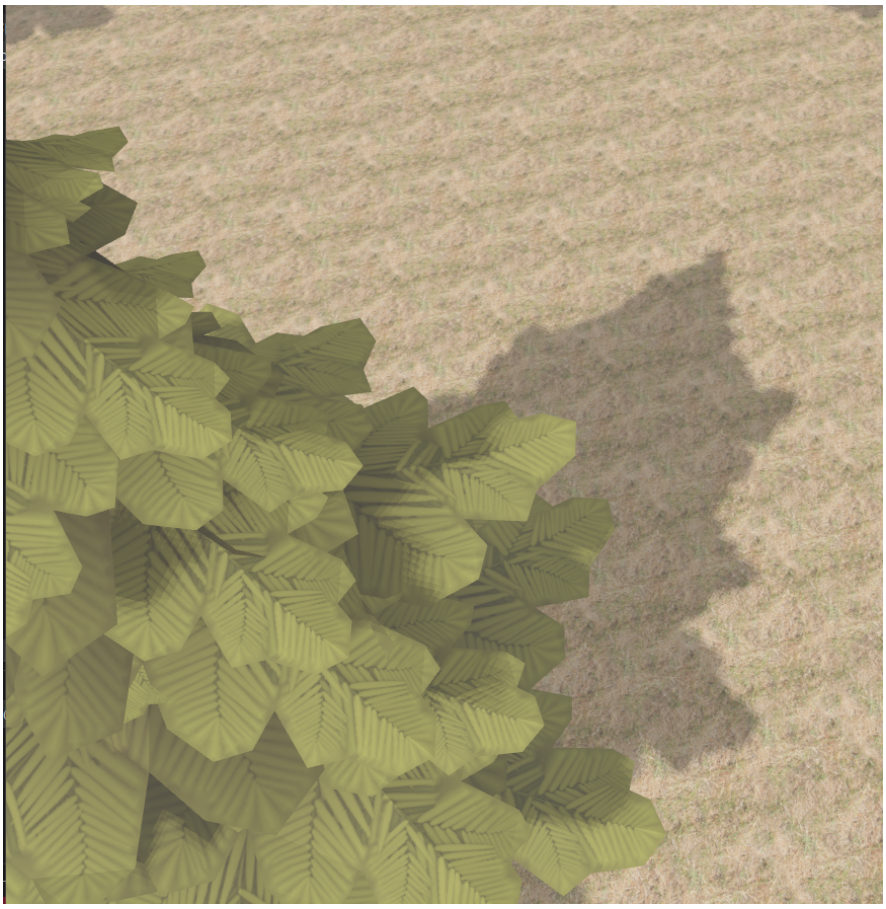

Figure 4: Shadow

```
7       view = myCamera.getViewMatrix();
8       glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(
            view));
9
10      glActiveTexture(GL_TEXTURE3);
11      glBindTexture(GL_TEXTURE_2D, depthMapTexture);
12      glUniform1i(glGetUniformLocation(myBasicShader.
            shaderProgram, "shadowMap"), 3);
13
14      glUniformMatrix4fv(glGetUniformLocation(myBasicShader.
            shaderProgram, "lightSpaceTrMatrix"),
15          1,
16          GL_FALSE,
17          glm::value_ptr(computeLightSpaceTrMatrix()));
18
```

```
    drawObjects(myBasicShader, false);
}
```

```
    void computeDirLight()
{
    //compute eye space coordinates
    vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
    vec3 normalEye = normalize(normalMatrix * fNormal);

    //normalize light direction
    vec3 lightDirN = vec3(normalize(view * vec4(lightDir,
        0.0f)));

    //compute view direction (in eye coordinates, the viewer
         is situated at the origin
    vec3 viewDir = normalize(- fPosEye.xyz);

    //compute ambient light
    ambient = ambientStrength * lightColor;

    //compute diffuse light
    diffuse = max(dot(normalEye, lightDirN), 0.0f) *
        lightColor;

    //compute specular light
    vec3 reflectDir = reflect(-lightDirN, normalEye);
    float specCoeff = pow(max(dot(viewDir, reflectDir), 0.0f
        ), 32);
    specular = specularStrength * specCoeff * lightColor;
}

vec3 ambientPoint;
float ambientStrengthPoint = 0.8f;
vec3 diffusePoint;
vec3 specularPoint;
float specularStrengthPoint = 0.1f;
float shininess = 100.0f;

void computePointLight(){
        //transform normal
    vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
        vec3 normalEye = normalize(fNormal);

        //compute light direction
        vec3 lightDirN = normalize(lightPos - fPosEye.xyz);

        //compute view direction
        vec3 viewDirN = normalize(cameraPosEye - fPosEye.xyz
            );
```

```
42
43          //compute ambient light
44          ambientPoint = ambientStrengthPoint * lightColor;
45
46          //compute diffuse light
47          diffusePoint = max(dot(normalEye, lightDirN), 0.0f)
                * lightColor;
48
49          //compute specular light
50          vec3 reflection = reflect(-lightDirN, normalEye);
51          float specCoeff = pow(max(dot(viewDirN, reflection),
                 0.0f), shininess);
52          specularPoint = specularStrengthPoint * specCoeff *
                lightColor;
53
54          float distance    = length(lightPos - fPosEye.xyz);
55          float attenuation = 1.0 / (constantAttenuation +
                linearAttenuation * distance +
                quadraticAttenuation * (distance * distance));
56
57          ambientPoint *= attenuation;
58          specularPoint *= attenuation;
59          diffusePoint *= attenuation;
60  }
61
62  vec3 ambientSpotLight;
63  float ambientStrengthSpotLight = 0.1f;
64  vec3 diffuseSpotLight;
65  vec3 specularSpotLight;
66  float specularStrengthSpotLight = 0.2f;
67
68  void computeSpotLight(){
69
70      vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
71      vec3 lightDirN = normalize(vec3(spotLightPos - fPosEye.
            xyz ));
72      float theta = dot(lightDirN, normalize(-spotLightDir));
73
74      if(theta > innerCutOff) {
75
76          ambientSpotLight = ambientStrengthSpotLight * vec3
                (0.4f,0.4f,0.4f);
77
78          vec3 normalEye = normalize(fNormal);
79          diffuseSpotLight = max(dot(normalEye, lightDirN),
                0.0f) * lightColor;
80
81          vec3 viewDirN = normalize(cameraPosEye - fPosEye.xyz
                );
82          vec3 reflection = reflect(-lightDirN, normalEye);
```

```
83          float specCoeff = pow(max(dot(viewDirN, reflection),
                0.0f), shininess);
84          specularSpotLight = specularStrengthPoint *
                specCoeff * lightColor;
85
86          float epsilon   = innerCutOff - outerCutOff;
87          float intensity = clamp((theta - outerCutOff) /
                epsilon, 0.0, 1.0);
88
89          float distance    = length(lightPos - fPosEye.xyz);
90              float attenuation = 1.0 / (constantAttenuation +
                      linearAttenuation * distance +
                    quadraticAttenuation * (distance * distance))
                    ;
91
92          specularSpotLight *= intensity;
93              diffuseSpotLight *= intensity;
94
95          ambientPoint *= attenuation;
96              specularPoint *= attenuation;
97              diffusePoint *= attenuation;
98      }
99      else{
100         ambientSpotLight = vec3(0.0f);
101         diffuseSpotLight = vec3(0.0f);
102         specularSpotLight = vec3(0.0f);
103     }
104 }
```

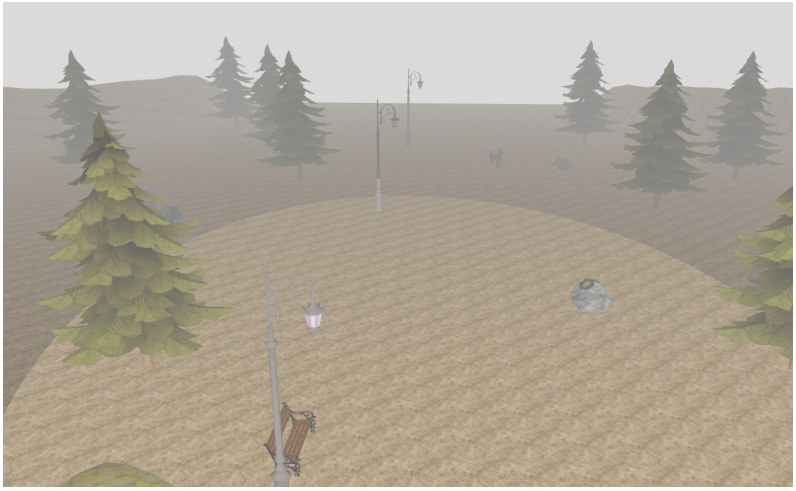

Figure 5: Spotlight

```
float computeFog()
{
   vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
   float fogDensity = 0.03f;
   float fragmentDistance = length(fPosEye);
   float fogFactor = exp(-pow(fragmentDistance * fogDensity
       ,2));

   return fogFactor;
}
    void main()
{
    computeDirLight();
    computePointLight();

    if(ifSpotLight==1){
        computeSpotLight();
    }

    ambient += ambientPoint;
    diffuse += diffusePoint;
    specular += specularPoint;

    ambient += ambientSpotLight;
    diffuse += diffuseSpotLight;
    specular += specularSpotLight;

    //compute final vertex color
    float shadow = computeShadow();
    vec3 color = min((ambient + (1.0f - shadow)*diffuse) *
        texture(diffuseTexture, fTexCoords).rgb + (1.0f -
        shadow)*specular * texture(specularTexture,
        fTexCoords).rgb, 1.0f);

    float fogFactor = computeFog();

    vec4 fogColor = vec4(0.5f,0.5f,0.5f,1.0f);
    fColor = vec4(color, 1.0f);
    fColor = (1 - fogFactor)*fogColor + fColor*fogFactor;

    float gamma = 2.2;
    fColor.rgb = pow(fColor.rgb,vec3(1.0/gamma));
}
```

Applied gamma correction to the final color and calculated the fog.

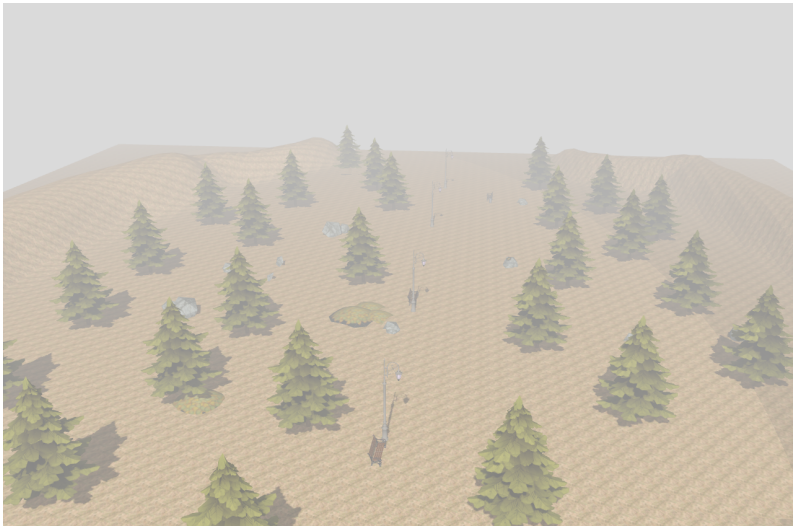

Figure 6: Scene From Above

4 User Guide

Use W,A,S,D keys to move around, V to disable the spotlight, N to start animation, B to rotate the directional light, M for changing the viewing,Q and E to rotate the scene and the mouse for rotating the camera.

5 Conclusions and Further Development

As a conclusion we can say that although the project is pretty simple it uses some techniques that improve the photorealism (such as the soft edge shadow algorithm, lights, shadows, movements). For Further Development there are a lot of things to add, making the point light and spot light work better, for shadow computation perhaps using something like Random Sampling to make the transition from shadow more natural, for light calculation I could have used the Blinn model of illumination instead of the phong model, for shadows I could have used bounding volumes, more effects can be added such as rain, thunder, fire, collision detection could be added using a Octrees or using the GJK algorithm.
All in all there are a lot of improvements that can be done to the project but for a first time application in OpenGL it looks pretty good. :)