



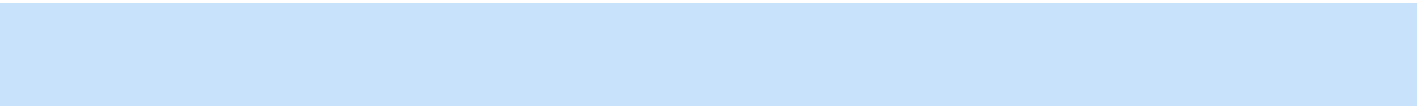
Graphical Processing Systems 2021-2022

Project Documentation

Chereji Iulia-Adela

Group 30434-1

14.01.2022



Contents

1	Subject specification	- 3 -
2	Scenario	- 3 -
2.1	Scene and objects description	- 3 -
2.2	Functionalities.....	- 6 -
3	Implementation details	- 7 -
3.1	Functions and special algorithms	- 7 -
3.1.1	Time passing and sun light dissappearing	- 7 -
3.1.2	Lamp positional light.....	- 8 -
3.1.3	Farmer's flashlight (spotlight type of source)	- 9 -
3.1.4	Camera animation.....	- 10 -
3.1.5	Windmill rotation	- 11 -
3.2	Graphics model	- 12 -
3.3	Data structures.....	- 13 -
3.4	Class hierarchy	- 13 -
4	Graphical user interface presentation / user manual	- 13 -
5	Conclusions and further developments	- 14 -
6	References.....	- 14 -

1 Subject specification

For this project I decided to do a small farm scene using OpenGL. The time passes in the scene so we can see it during the day and during the night. With the background and the grass on the ground I wanted to make it look like a summer day. I took the farmhouse object from one of this semester's laboratory resources and I kept building the scene around it so that's how it became a farm.

2 Scenario

The scene looks like a courtyard next to a stable. There are a two bulls that stay next to a few trees in the shadow. There is a lamp in the left side with a light sensor that turns on in the evening and off in the morning, and a windmill that powers the lamp. The farmer can walk around the closed yard and he has a flashlight for when he checks on the animals at night time. The scene is surrounded by a long stall and a fence. Sometimes there is fog in the area but not very thick.

2.1 SCENE AND OBJECTS DESCRIPTION



The scene represents a summer day at the farm. The background is a Skybox object with the image of some trees and a blue house. The ground is a square 3D object on which I mapped a grass texture and there are in total 4 squares in the scene.

All the objects that can be found in the scene are .obj files with .mtl texture files associated. The .mtl files map the objects coordinates to the ones of some pictures (.jpg or .png) The objects are:

- The grass is made out of 4 ground objects that I took from one of the laboratories' resources but I replaced the texture image with a green grass image.
- The sky is a Skybox object on which I mapped 6 images of a skybox I found on the internet.



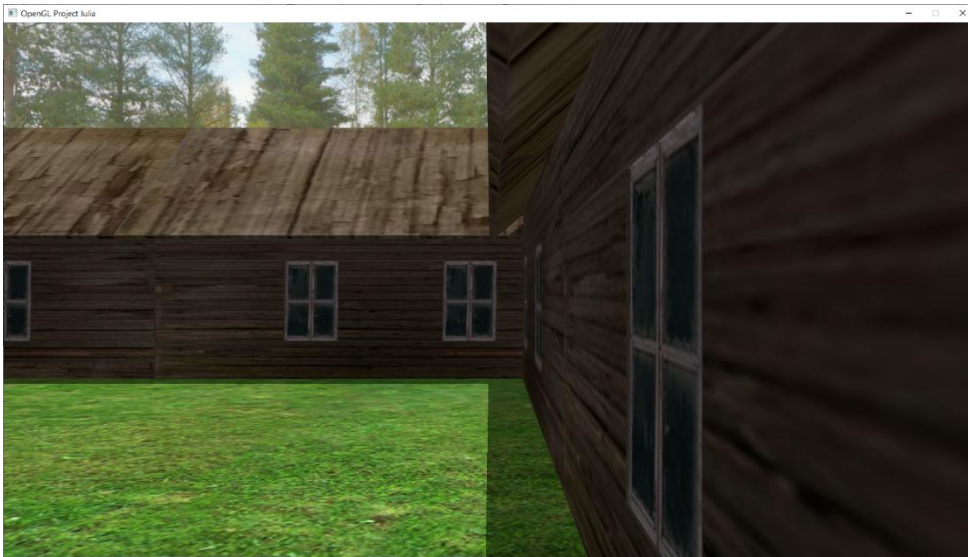
- The 2 bulls are the same object, drawn twice. I downloaded all of their components (.obj, .mtl, .png). They have smooth surfaces.
- The street lamp is also downloaded from the internet. And I made it seem like a source of light.
- The trees are downloaded from the internet and they are the most complex object in the scene. They have a lot of details and if I add more the application runs very slow.
- The windmill's body and it's airscrew are downloaded as 2 different objects. They used to have another color but I put the image of a rock wall on it. The reason for which I chose to



have 2 different objects is that I wanted to make the airscrew rotate.



- The stall is composed of more farmhouse objects that I took from the laboratory resources and put them next to each other to give the impression of a long surrounding stall.
- The fence is downloaded from the internet but it originally had no .mtl texture so the program could only draw it black. But I created a .mtl file and I downloaded a wood image to map onto it. Then I put more fence objects next to each other to create a longer fence.
- The light cube which appears only when the ‘C’ key is pressed to show the position of the directional light.



2.2 FUNCTIONALITIES

- Visualization of the scene: all the objects in the scene are arranged and placed using translation, scale and rotation.
- Camera movement: the camera moves in the scene in 4 directions using keyboard input and it can also rotate on 2 axis using the keyboard. The movement is restricted to a specified area so the camera can not get out of the scene on x and z axis. But it can go up and down. The camera also moves through animation. It moves forward until it reaches the border. At that moment it rotates until it can move freely.
- Specification of light sources: 3 light sources were specified, directional light (sun), positional light (lamp) and spotlight (flashlight of the farmer).
- Viewing the surfaces: The objects can be seen as solid things (you can not see through them) or wireframe (only edges are seen). In the scene there are polygonal surfaces (ground, leaves of trees, the stall, the windmill) and smooth surfaces (bulls, lamp).
- Texture mapping and materials: All objects have mapped textures. Some of them I mapped and created myself, some I downloaded with all the components.
- Shadows: The shadows were computed using the algorithm presented in the laboratory, using the directional light.

- Animation of object components: The airscrew of the windmill rotates continuously at a low speed to simulate the wind blowing on it.
- Photo-realism: the scene can contain fog (made with the algorithm from the laboratory).

3 Implementation details

The following chapter will describe the algorithms that I used for the more complicated details of the project.

3.1 FUNCTIONS AND SPECIAL ALGORITHMS

3.1.1 TIME PASSING AND SUN LIGHT DISSAPPEARING

I wanted to simulate the night and day effect. For this I had to reduce the light intensity of the directional light from maximum to zero and then do it in reverse, in a loop.

3.1.1.1 Possible solutions

The solution that I chose is to have a function that I call from the application loop:

```
void processTimePassing() {
    if (stayLightOrDark) {
        if (timeLightOrDark <= 0.0f) {
            stayLightOrDark = false;
        }
        else timeLightOrDark -= timeSpeed;
    }
    else {
        if (increaseLight) {
            if (timeOfDay >= 1.0f) {
                increaseLight = false;
                stayLightOrDark = true;
                timeLightOrDark = 200 * timeSpeed;
            }
            else {
                timeOfDay += timeSpeed;
            }
        }
        else {
            if (timeOfDay <= 0.00f) {
                increaseLight = true;
                stayLightOrDark = true;
                timeLightOrDark = 200 * timeSpeed;
            }
            else {
                timeOfDay -= timeSpeed;
            }
        }
    }
}
```

The stayLightOrDark is a variable that I had to introduce to make the scene stay on total darkness and total light for a period of time because otherwise it was not realistic. The timeSpeed can be modified by user input to have the scene stay as it is for more or make it go fast to see the changes. The timeOfDay computation is the actual purpose of this method. This variable takes values between 0 and 1 and is passed to the fragment shader. In the shader it is used when calculating the ambient, diffuse and specular components of the directional (sun) light, to “turn on” the lamp if it is dark, to adjust the shadow strength and the fog (if I do not use it here, at night the scene with fog is light gray because only the fog is seen).

```
ambient = ambientStrength * lightColor * timeOfDay;
```

```
diffuse = max(dot(normalEye, lightDirN), 0.0f) * lightColor * timeOfDay;
```

```
specular = specularStrength * specCoeff * lightColor * timeOfDay;
```

```
if(timeOfDay<=0.05f)
```



```

computeDirLamp();

shadowSun*=timeOfDay;

fColor = mix(fogColor*timeOfDay, vec4(color, 1.0f), fogFactor);

```

3.1.1.2 *The motivation of the chosen approach*

I thought of this approach myself and I managed to implement it very easily so I kept it.

3.1.2 LAMP POSITIONAL LIGHT

I had to make the lamp give just enough light so only a few objects can be seen so I spent a lot of time trying values for the constants determining the attenuation of the light.

3.1.2.1 *Possible solutions*

I sent the lamp position as a uniform to the vertex shader from the main application and I made the light color yellow. All the algorithm is in the shader:

```

float constant = 1.0f;

float linear = 0.22f;

float quadratic = 0.20f;

void computeDirLamp()
{
    //compute eye space coordinates

    vec4 fPosEye = view * model * vec4(fPosition, 1.0f);

    vec4 lampPosEye = view * vec4(lampPos, 1.0f);

    vec3 normalEye = normalize(normalMatrix * fNormal);

    vec3 yellowColor = vec3(1.0f, 1.0f, 0.0f);

    //normalize light direction

    vec3 lightDirN = vec3(normalize(lampPosEye.xyz - fPosEye.xyz));

    //compute distance to light

    float dist = length(lampPosEye.xyz - fPosEye.xyz);

    //compute attenuation

    float att = 1.0f / (constant + linear * dist + quadratic * (dist * dist));

    //compute view direction

    vec3 viewDir = normalize(- fPosEye.xyz);

    //if(dist < 11.0f){

    //compute ambient light

```



```

vec3 ambient2 = att * ambientStrength * yellowColor;

ambient += ambient2;

//compute diffuse light
vec3 diffuse2 = att * max(dot(normalEye, lightDirN), 0.0f) * yellowColor;
diffuse = max(diffuse, diffuse2);

vec3 reflectDir = reflect(-lightDirN, normalEye);

float specCoeff = pow(max(dot(viewDir, reflectDir), 0.0f), shininess);

vec3 specular2 = att * specularStrength * specCoeff * yellowColor;

specular = max(specular, specular2);

//}
}

```

3.1.2.2 *The motivation of the chosen approach*

It is the method used in the laboratory and it worked perfectly.

3.1.3 FARMER'S FLASHLIGHT (SPOTLIGHT TYPE OF SOURCE)

I wanted to give the farmer a flashlight so he can see around in the dark. For this I encountered a lot of problems.

3.1.3.1 *Possible solutions*

In the main application I send the Boolean variable flashLightOn to the fragment shader to know when to compute this light. I made it yellow also. All the computations are made in the shader:

```

void computeDirFlashlight()
{
    float cutOff = cos(radians( 12.5f ));

    float outerCutOff = cos( radians( 17.5f ));

    //compute eye space coordinates
    vec4 fPosEye = view * model * vec4(fPosition, 1.0f);
    vec3 normalEye = normalize(normalMatrix * fNormal);
    vec3 yellowColor = vec3(1.0f, 1.0f, 0.0f);

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

    //compute view direction
    vec3 viewDir = normalize(- fPosEye.xyz);

    vec4 camFrontDirectionEye = view * vec4(cameraFrontDirection, 0.0f);

```

```

float dist = length(- fPosEye.xyz);

float strength = 3/dist;

float theta = dot(lightDirN, normalize(-camFrontDirectionEye.xyz));

float epsilon = (cutOff - outerCutOff);

float intensity = clamp((theta - outerCutOff) / epsilon, 0.0, 1.0);

ambient += ambientStrength * yellowColor* intensity * strength;

//compute diffuse light

diffuse += max(dot(normalEye, lightDirN), 0.0f) * yellowColor * intensity * strength;

//compute specular light

vec3 reflectDir = reflect(-lightDirN, normalEye);

float specCoeff = pow(max(dot(viewDir, reflectDir), 0.0f), shininess);

specular += specularStrength * specCoeff * yellowColor * intensity * strength;

//diffuse += intensity* yellowColor;

//specular += intensity* yellowColor;

}

```

The cutOff, outerCutOff, theta, epsilon and intensity computations I took from a YouTube video which I will link in the references. But for the theta I had to do a modification: I needed the camera front direction to calculate the light's direction so I had to get it in the main application and always send it to the shader. The strength is added by me to get the effect that I wanted.

3.1.3.2 *The motivation of the chosen approach*

I found this algorithm and even though it was a lot of work I finally got it to work and I did not try another method.

3.1.4 CAMERA ANIMATION

I wanted to make the camera (farmer) move alone in the scene but I did not want it to go out or to get stuck on the edge. I also did not want to make it rotate 180 degrees instantly because that would not have been realistic. I wanted a smooth turn, and not always by the same angle so we don't explore the same path always. Also when the animation is on we can not manually move or rotate the camera.

3.1.4.1 *Possible solutions*

I chose to have this code in the main application loop:

```

if (animationOn) {
    if (rotate)
        rotateAtCollision();
    else
        CheckForColisionAndMove(gps::MOVE_FORWARD, cameraSpeed);
}

```

```

else
    processMovement();

```

The animationOn variable is modified by user input. The rotate variable is set to true when we collide with a boundary and we have to rotate and set to false when we rotated enough so that we are unstuck and can move forward. The CheckForCollisionAndMove method is also called when we manually move the camera and it checks if we try to move into an unpermitted area, and if so, and if the animation is on, it starts the rotation process.

```

void CheckForCollisionAndMove(gps::MOVE_DIRECTION direction, float speed) {
    gps::Camera newCamera(
        myCamera.getCameraPosition(),
        myCamera.getCameraTarget(),
        myCamera.getCameraUpDirection());
    newCamera.move(direction, speed);
    float newx = newCamera.getCameraPosition().x;
    float newz = newCamera.getCameraPosition().z;
    if (newx <= -19 || newx >= 19 || newz <= -19 || newz >= 19) //out
    {
        if (animationOn) {
            rotate = true;
            totalTimesToRotate = 25 + (std::rand() % (180 - 25 + 1));
            currentRotationTimes = 0;
        }
        else return;
    }
    else {
        myCamera.move(direction, speed);
        view = myCamera.getViewMatrix();
        return;
    }
}

```

The method creates a copy of the camera and moves it where we try to move then it checks if the new camera is outside the allowed zone. If so, and if we aren't into the animation it just does not call the camera move method so we can't advance. If we are in the animation mode and we try to go outside the allowed area it starts the rotation process: makes rotate true, generates a random number for the totalTimesToRotate variable (so we get a random angle), and makes currentRotationTimes equal to zero. If we want to move to a valid area, even if we are animated or not, it just makes the movement.

```

void rotateAtCollision() {
    if (currentRotationTimes >= totalTimesToRotate)
        rotate = false;
    else {
        myCamera.rotate(0, cameraRotationAngle);
        currentRotationTimes++;
    }
}

```

The rotateAtCollision method is called from the application main loop and performs the rotation by a small angle at each loop iteration until the previously generated random number of rotations are performed. I did it like this to achieve a smooth rotation. After all rotations are performed it sets rotate to false and the main loop checks for collision again (because it is possible that the random angle did not get us out of trouble and we might still try to escape the scene, so we shouldn't just move).

3.1.4.2 *The motivation of the chosen approach*

I am pleased with the result that I got using this approach so I kept it.

3.1.5 WINDMILL ROTATION

I wanted to make the windmill's airscrew rotate slowly to mimic a real windmill.

3.1.5.1 *Possible solutions*

For this I solved the problem in the rendering of the windmill method:


```

angle_windmill += angle_windmill_increment;
if (angle_windmill >= 360)
    angle_windmill = 0;

modelWindmill2 = glm::scale(modelWindmill2, glm::vec3(0.1f, 0.1f, 0.1f));
modelWindmill2 = glm::translate(modelWindmill2, glm::vec3(76.2f, 17.5f, -93.0f));
modelWindmill2 = glm::rotate(modelWindmill2, glm::radians(angle_windmill), glm::vec3(0.0f, 0.0f, 1.0f));

```

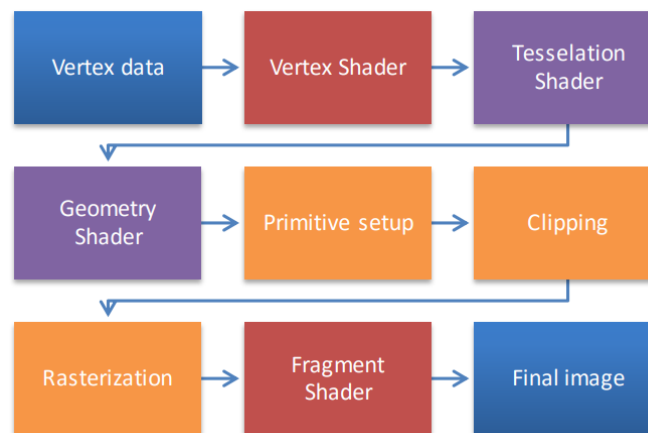
The `angle_windmill` and `angle_windmill_increment` are global variables. The former one is incremented by the second at every iteration of the application main loop, when the rendering is done. If it reaches 360 I make it 0 so it doesn't overflow after a few minutes and make the app crash. Then I do the rotation after the scale and the translation so the result is the correct one (this means that the rotation will be performed first, when the object is still at the origin). I had to try a lot of values for the increment until I got the speed of rotation that I wanted.

3.1.5.2 The motivation of the chosen approach

This is the way we learned at the course last year that objects should be rotated so this is what I used.

3.2 GRAPHICS MODEL

The Graphics model used in the implementation is the one defined by the OpenGL.

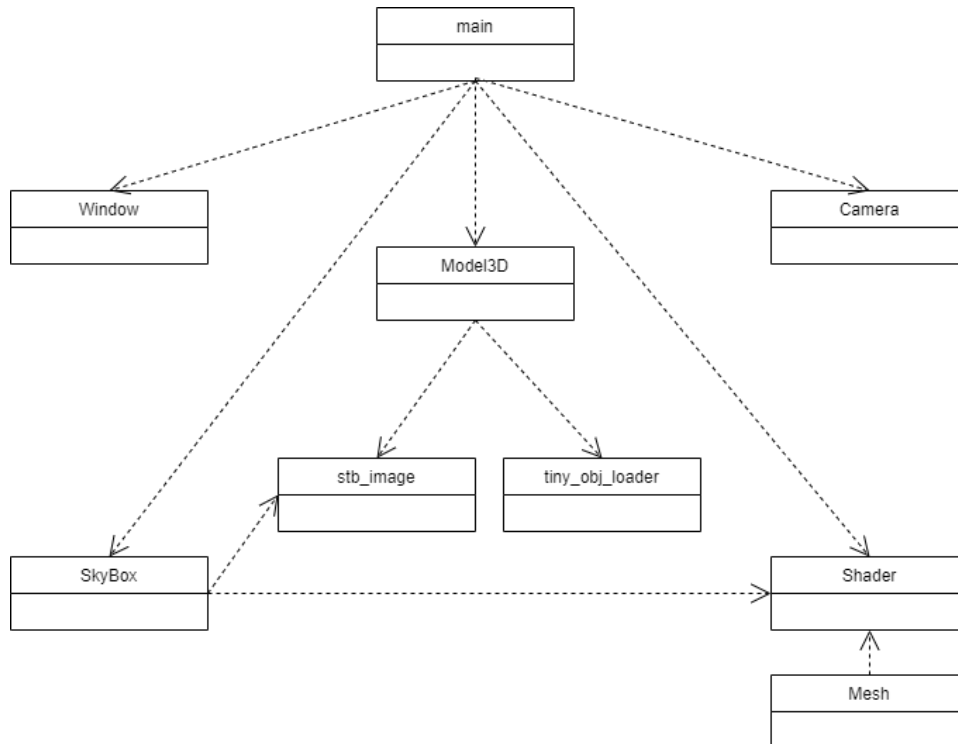


I did not program the Tasselation and Geometry Shaders, only the Vertex and Fragment Shaders. In the Fragment shader I did most of the calculations: directional positional and spot light, fog, shadows, textures and colors. In the Vertex shader I used the light space transformation matrix in order to compute the current vertex's position in the light space (scene viewed from the light sources perspective) so I can get its z value since the values from the depth map were in light's coordinates and I needed to make a comparison to compute the shadow value.

3.3 DATA STRUCTURES

The only data structures that I used were the ones provided to us in the project template. The only one that I modified is the Camera class in which I implemented the move and rotate methods and I added some getters for camera position, target and front direction that I needed to make a copy of the camera for collision checking and also I needed the camera front direction for the flashlight computation. I also added a preview function to see the whole scene.

3.4 CLASS HIERARCHY



4 Graphical user interface presentation / user manual

The user can interact with the application only using the keyboard:

- Use arrows to move the camera Front, Back, Right, Left
- +- to make the time go faster or slower
- N, M, Y, H to rotate the camera Left, Right, Up, Down
- F to activate the fog
- L to activate the flashlight
- W to see wireframe
- C to see the light cube
- A to start the animation
- P to see the position of the camera in the console
- S to start the preview
- ESC to stop the application

5 Conclusions and further developments

This project turned out to be really interesting and I am glad I did it. Further developments that could be made:

- The shadows could take into considerations all light sources
- More objects can be added
- Real collision detection algorithms could be implemented
- The scene could be made bigger

6 References

The laboratory works

https://www.glfw.org/docs/3.3/group__window.html

<https://www.khronos.org/registry/OpenGL-Refpages/>

<https://www.glfw.org/docs/latest/quick.html>

https://www.glfw.org/docs/3.3/group__window.html

<https://learnopengl.com/Advanced-OpenGL/Depth-testing>

<https://learnopengl.com/Advanced-OpenGL/Face-culling>

<https://stackoverflow.com/questions/21830340/understanding-glmlookat>

<https://www.youtube.com/watch?v=Ut6poChkSjA>

<https://www.youtube.com/watch?v=tmCOMzAA4rc>

<https://stackoverflow.com/questions/7560114/random-number-c-in-some-range>

<https://community.khronos.org/t/color-tables/22518/2>

<https://stackoverflow.com/questions/33690186/opengl-bool-uniform>

https://www.glfw.org/docs/3.3/group__keys.html

<https://www.cgtrader.com/3d-models>

<https://free3d.com/>

<https://stackoverflow.com/questions/16578027/rotating-an-object-around-a-fixed-point-in-opengl>

<https://www.turbosquid.com/>