**OpenGL Project Documentation**

Ardelean Eugen Richard

30433

1. Contents  
2. Subject specification  
3. Scenario  
    3.1. Scene and objects description  
    3.2. Functionalities  
4. Implementation details  
    4.1. Functions and special algorithms  
           4.1.1. Possible solutions  
           4.1.2. The motivation of the chosen approach  
    4.2. Graphics model  
    4.3. Data structures

    4.4. Class hierarchy  
5. Graphical user interface presentation / user manual  
6. Conclusions and further developments  
7. References

**1. Contents**The application contains a 3D environment containing several objects, presenting a winter-themed scenario.

I created a Medieval Fantasy scene, where the user can interact with certain objects. For the purpose of defeating his enemy, an ogre. The user has to discover objects, that he can interact with, and explore the scene in order to win the game.

For creating this project Microsoft Visual Studio 2017 with several libraries was used.

**2. Subject specification**The specifications of this project contain:

* visualization of the scene: scaling, translation, rotation, camera movement
  + using keyboard and mouse
  + using animation
* specification of light sources (minimum two different lights)
* viewing solid, wireframe objects, polygonal and smooth surfaces
* texture mapping and materials
  + textures quality and level of detail
  + textures mapping on objects
* exemplify shadow computation
* exemplify animation of object components
* photo-realism, scene complexity, detailed modeling, algorithms development and implementation (objects generation, collision detection, shadow generation, fog, rain, wind), animation quality, different types of light sources (global, local, spotlights)
* the documentation

Of these, I have managed to implement all but the shadow computation, of which code I have added but they don’t appear.

**3. Scenario  
    3.1. Scene and objects description**

The scene is a winter one, with a white ground object and a winter-themed skybox, it also has some snowflakes falling from the sky, to help immerse the user into the scene.

There are several other objects:

One of the buildings, the watchtower, looks like this:



Another significant building in the scene is the windmill:



There is also a box, created with vertices and triangles using the following texture on each face:



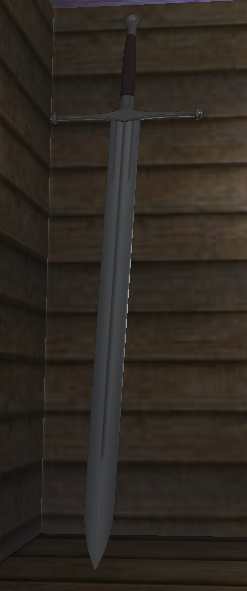
Also there are walls of the following object that define the scene:



The main antagonist for this game looks like this:

****

And as weapons for defeating this formidable antagonist we have the valyrian steel sword of Ned Stark, Ice, and mighty hammer of Thor, Mjolnir:



For the snow, each snowflake object looks like this:

And just because no fantasy setup is complete without a dragon, I added one to the scene:



By using the mouse and keyboard, you can move around to view the scene.

**3.2. Functionalities**

The user move around the scene using the mouse and keyboard, and activate camera animation to present the whole scene.

He or she can also interact with certain objects(box, weapons and characters), he can push a box, he can pick up weapons from around the scene and switch between them, and of course use them to attack. In case of extreme darkness the player can activate his flashlight which will light the way in front of him.

**4. Implementation details  
    4.1. Functions and special algorithms**

Unfortunately, there are no special algorithms in my project.

The most special algorithm used in the project is for generating 300 snowflakes with random location to fall from the sky and each reset their starting position when touching the ground.

**4.1.1. Possible solutions**double random(int min, int max)

{

return (double)rand() / (RAND\_MAX + 1) \* (min - max) + min;

}

glm::vec3 snow[300];

for (int i = 0; i < 300; i++)

{

snow[i] = glm::vec3(10.0f+random(0, 2000) / 100, snowY+ random(0, 6000)/100, 7.5f+random(0, 2000) / 100);

}

while (!glfwWindowShouldClose(window))

{

…

for (int i = 0; i < 300; i++)

{

snow[i].y = snow[i].y - 0.5f;

model = glm::mat4();

if (snow[i].y < -1.75f)

snow[i].y = snow[i].y + 60.0f;

model = glm::translate(model, snow[i]);

…

modelShader.setMat4("model", model);

snowOBJ.Draw(modelShader);

}

…

}

**4.1.2. The motivation of the chosen approach**I used a function that generates a random number between 2 given arguments based on the rand() function that generates a pseudo-random number between 0 and 1.

We take a vector of vec3 variables and we initialize a certain number (I chose 300 based on the speed with which the application was running) of them. The snowflakes will be generated on a 20x20x60 space. Because the generated number is a integer, I chose to generate a number up to 10,000 and then divide it by 100 so it will have 2 decimals, this will help with realism because there is a smaller of chance of overlapping snowflakes without actually having to check for overlapping which would have a significant overhead.

The third part which is in the while, will be rendered along with all other objects, that is why there is a for that decreases the value of the y axis of each snowflake by 0.5, because this will be decreased each time the objects are rendered and it will give the illusion of falling.

When one of these snowflakes get under a certain value on the y axis it will be reset by adding 60 to the y axis. Each of these modifications have to be added to the model by translations, along with the rotations and the scaling that you need for aesthetics. Then we send to the shader the model and draw the snowflake object.

Some other worthy mentions, are the positioning of the weapons and the feature of attacking.

float attackAngle = 0.01f;

bool attackBack = false;

…

if (attack == true && ((hammer == true && sword == false) || selectHammer == true))

{

attackAngle = attackAngle + 0.05f;

if (attackAngle > 0.90f)

attackBack = true;

if (attackBack == true)

attackAngle = attackAngle - 0.10f;

if (attackAngle < 0.0f)

{

attackBack = false;

attack = false;

}

…

Rotation by AttackAngle

}

The bool variable attack becomes true when alt key is pressed, if you have a weapon.

The idea is that when the attack button is pressed the weapon starts lowering the weapon, by decreasing its angle. until it hits a certain angle, here 90 degrees, this will make the bool attackBack true, which will decrease the angle, when it hits 0.00f, we know the attack has finished to we set both attack and attackBack to false.

And the setting of weapon position:

float weaponPosAngle = 0.0f;

…

if (camera.Yaw == -45.0f || camera.Yaw == 135.0f)

weaponPosAngle = camera.Yaw;

else

weaponPosAngle = -90 - camera.Yaw;

…

And then rotate the weapon by weaponPosAngle according to the Y axis and set the position of the Z axis slightly in front of the camera to be able to see the object.

These are the most special algorithms I have used.

**4.2. Graphics model**

A scene is made of objects, which are made of meshes, which are made of polygons, which are made of vertexes. Practically, everything is made of triangles, because it is guaranteed to be a convex polygon, so the computations are easier to make. This can be seen in wireframe mode which can be observed on the antagonist by pressing the Y key.

**4.3. Data structures**

The main data structures used are vectors and matrices.

Vectors can represent: normal, positions, coordinates, colors…

Other data structures, can be found with the 3 types of lights: directional lights, point lights and spotlights.

For each type of light, a structure was defined.

Any light has the ambient, diffuse and specular components.

For point lights, we want them to fade corresponding to distance. For this we use the float values of constant, linear and quadratic to create attenuation of the light according to how far something is from the source. Position represents the position of the object that gives the light.

struct PointLight {

vec3 position;

vec3 ambient;

vec3 diffuse;

vec3 specular;

float constant;

float linear;

float quadratic;

};

Directional light represents light waves that are parallel, that come from an infinite distance (ex. the sun) so there is no position for it but there is a direction, which is valid for any of the light waves as vectors are.

struct DirLight {

vec3 direction;

vec3 ambient;

vec3 diffuse;

vec3 specular;

};

Spotlights are lights that light only in one specific direction instead of all around them, (ex. flashlight), so a position of this light, the direction in which it lights and a cutoff angle (for the radius of the cone of light), the outercutoff is a value that is used to smooth edges of the flashlight by creating an inner and outer cone (light fades from inner to outer cone).

struct SpotLight {

vec3 position;

vec3 direction;

float cutOff;

float outerCutOff;

float constant;

float linear;

float quadratic;

vec3 ambient;

vec3 diffuse;

vec3 specular;

};

**4.4. Class hierarchy**Additional classes:

Shader class:

- makes it easier to send information between app and shaders

Camera class:

- we practically move all objects in the scene in the reverse direction of our movement to create an illusion

Mesh class:

- a mesh is a single drawable entity (most objects are composed of multiple parts, ex: head, torso, limbs, a mesh is one of those parts)

Model class:

- a model is a combination of multiple meshes, or even objects

- uses mesh class

Shader class is used for loading and compiling the shaders.

Camera class is used for creating the illusion of a camera.

Model class used for loading the objects.

**5. Graphical user interface presentation / user manual**-close the window by pressing escape.

-move around the scene with the W, A, S, D keys and the mouse movement.

-zooming with mouse scroll

-activate snow by pressing R

-camera animation by pressing T

-viewing the antagonist in wireframe by pressing Y

-activating flashlight by pressing F

-interacting with the box by pressing E when in proximity

-grabbing object by pressing G when in proximity

-attacking by pressing LEFT ALT button if you have a weapon

**6. Conclusions and further developments**The scene could be more complex, with many more objects (characters and buildings) and animations. There could be many more interactions between user and characters.

The collision could be done more efficiently by using bounding boxes.

The snow could be made to not go through buildings but stop when it touches them, from where it could be translated to the top and start falling again. And it certainly can be implemented in a more efficient manner.

**7. References**

<https://learnopengl.com/>

<https://free3d.com/>

<https://sketchfab.com/>

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