Student: Octavian-Mihai Matei

Group: 3043

# OpenGL project C++

# Table of contents

QR to the github repository

# Specifications

The subject of the project consists in the photorealistic presentation of 3D objects using OpenGL library. The user directly manipulates by mouse and keyboard inputs the scene of objects. For this purpose, an ancient scene was created where the user can interact with the camera, move around the town, day-night cycle and many more functionalities.

# Scenario

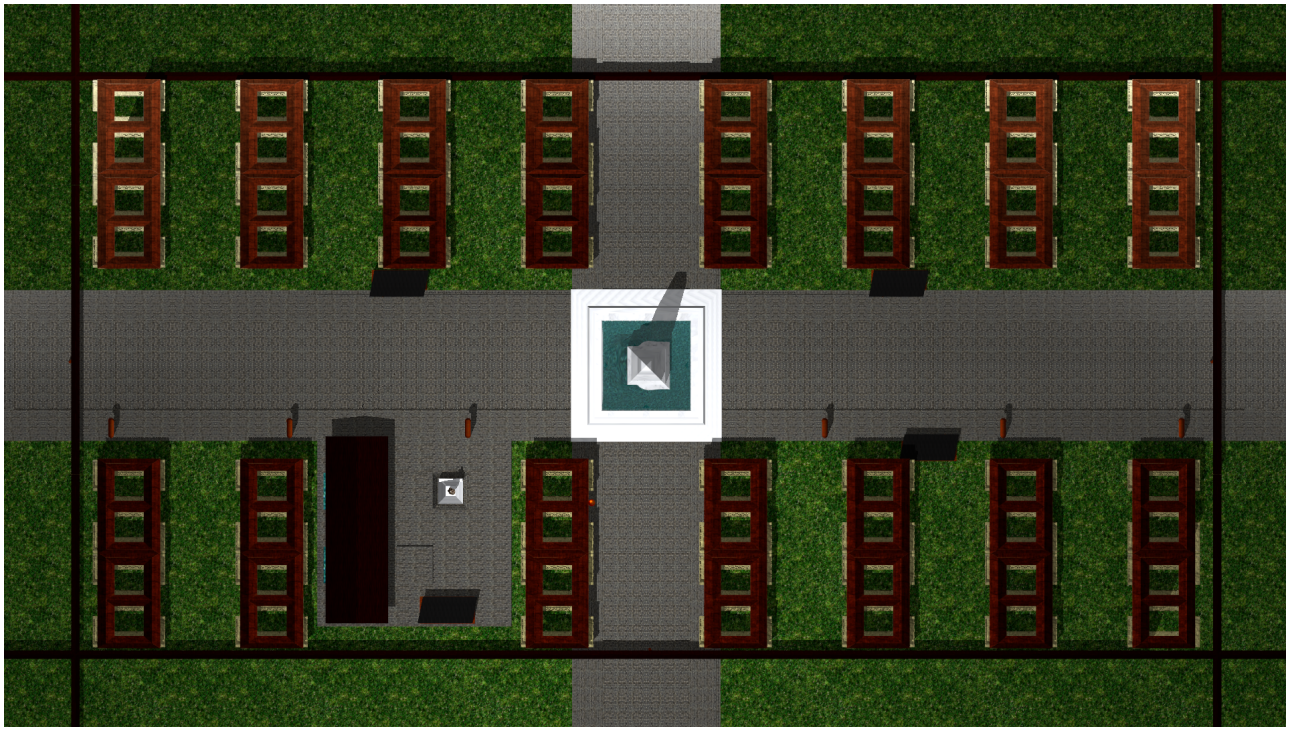
## Scene and objects description

The scene is composed of a town with buildings, a town obelisk square, a statue plazza, some stalls, street lights and city walls. In order to make the city "alive", the environment depicts a quent ancient/medieval city with ambiental city sounds, water splashing sound effects and merchants sounds around the stalls. The town has the following elements:

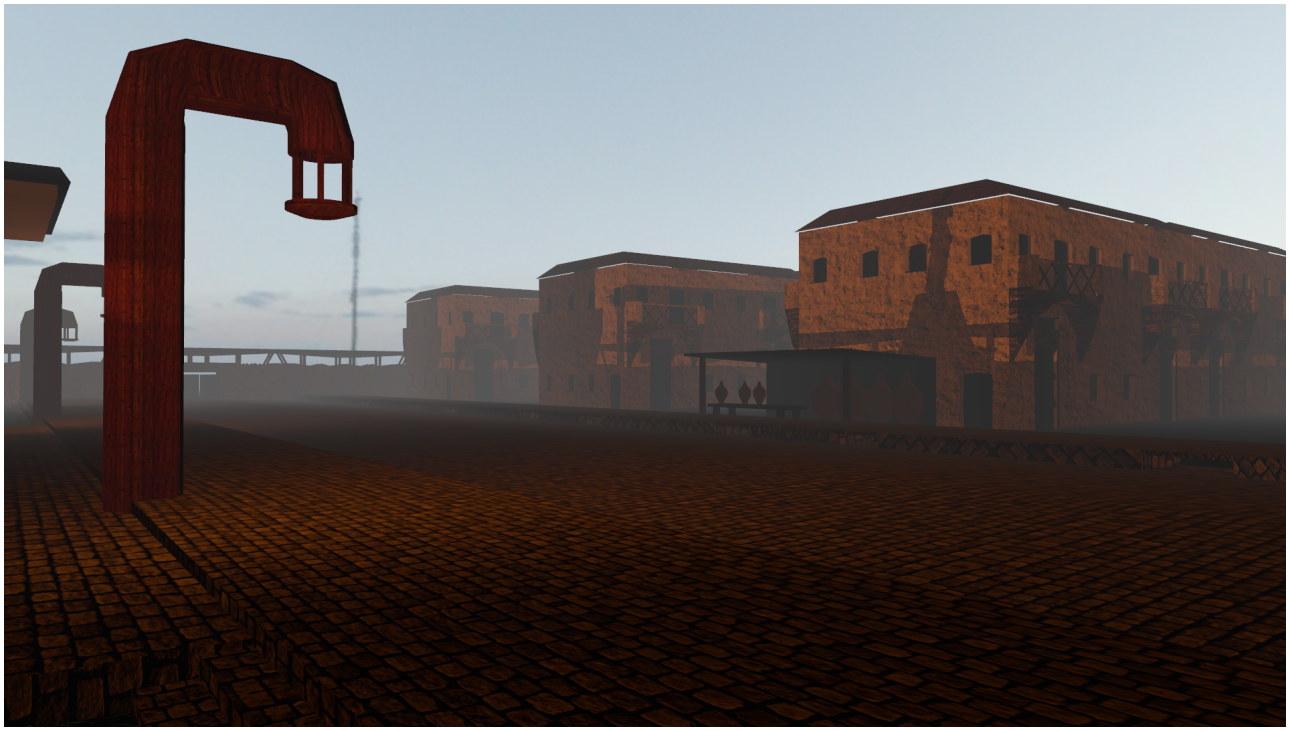
- City Walls
- City Gate Walls
- Streets
- Street Lamps with dynamic lighting
- A monument at the center of it
- A city forum where the people of the city could meet and trade
- Merchant stalls
- Grass
- Skybox

These elements were combined in order to produce the following demo images:

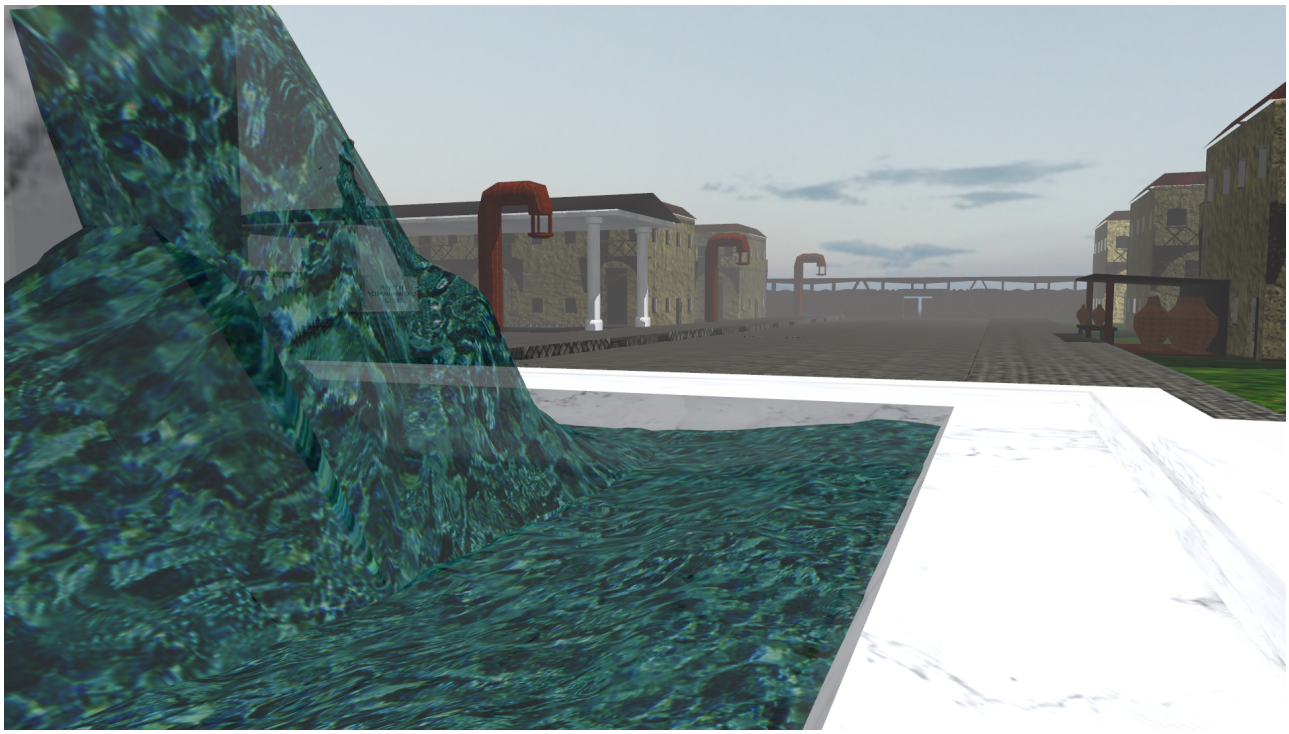
- Map view of the city



- Night time atmosphere



- View of the forum



# Functionalities

The player has many ways to control the world in this project. A clear way to see these controls is to start the game, because they are written in the cmd before the game opens GLWindow. For the ease of explanation, these are:

- WASD for movement
- Mouse to look around
- Scroll to change FOV
- F fullscreen/windowed mode
- M Map mode
- K walking/fly mode
- N autoday on/off
- if autoday off then:
    - - decrease time of day
    - + increase time of day
- , decrease movement speed
- . increase movement speed
- 1 turn on flashlight
- 2 turn off flashlight
- 7 GL_LINE vision mode
- 8 GL_POINT vision mode
- 9 GL_FILL vision mode

A small animation will play everytime the user enters or exists the map mode. In the map mode, the movement of the camera is blocked, the mouse events are disabled, the user cannot switch between walking and fly mode and the player cannot turn off/on the flashlight.

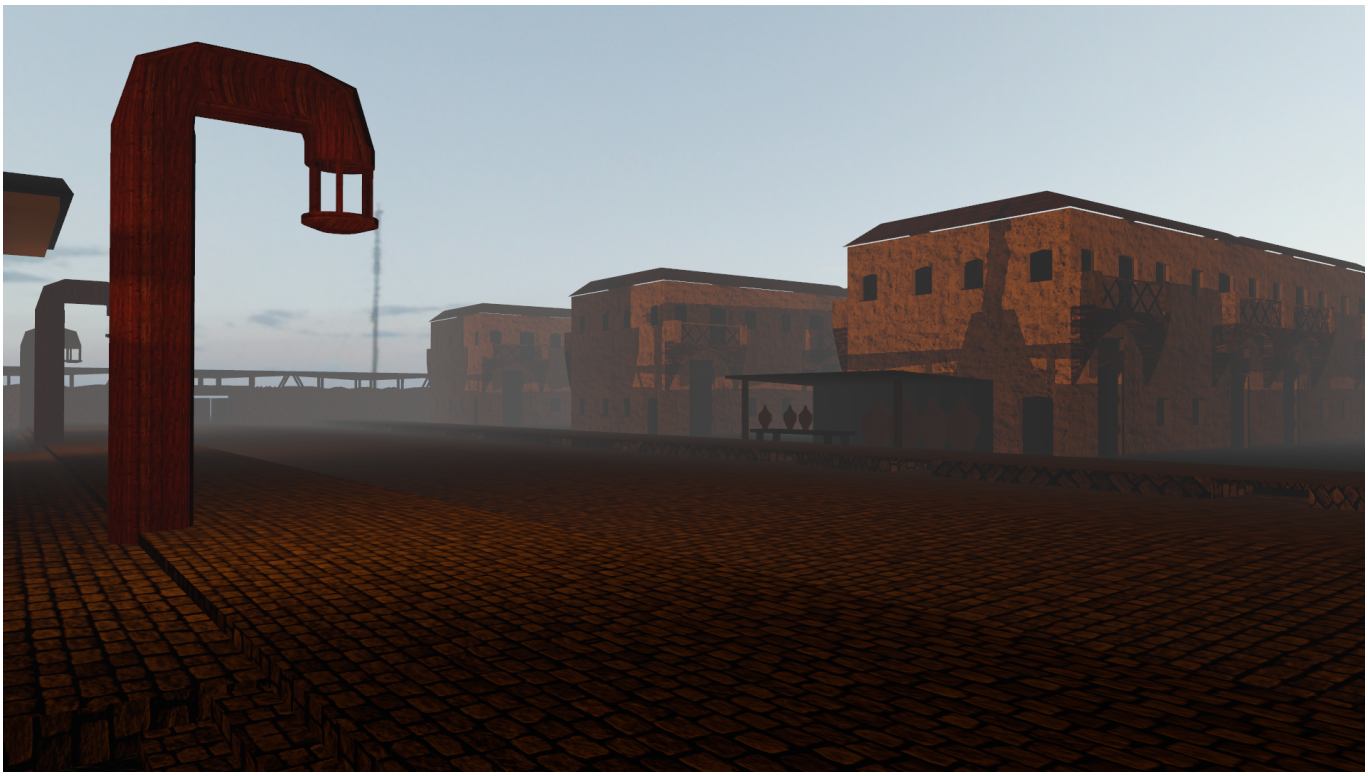
# Implementation details

## Functions and special algorithms

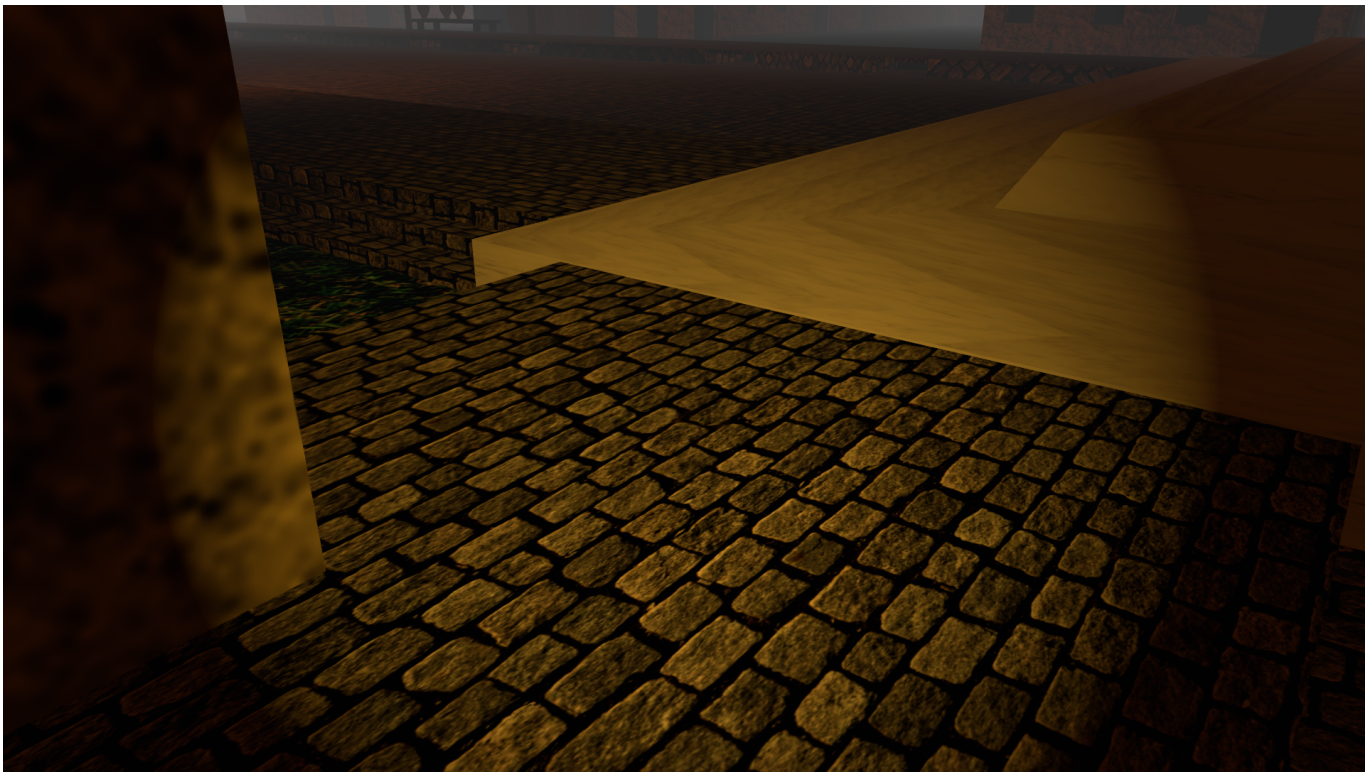
A range of algorithms were used to create the atmosphere and the look of the application:

1. Point light
2. Spotlight light
3. Dynamic Percentage-closer filtering
4. Water movement
5. Day-Night cycle with moving Sun and responsive street lights
6. Map Animation
7. Music and positional sounds

### Pointlight



## Spotlight



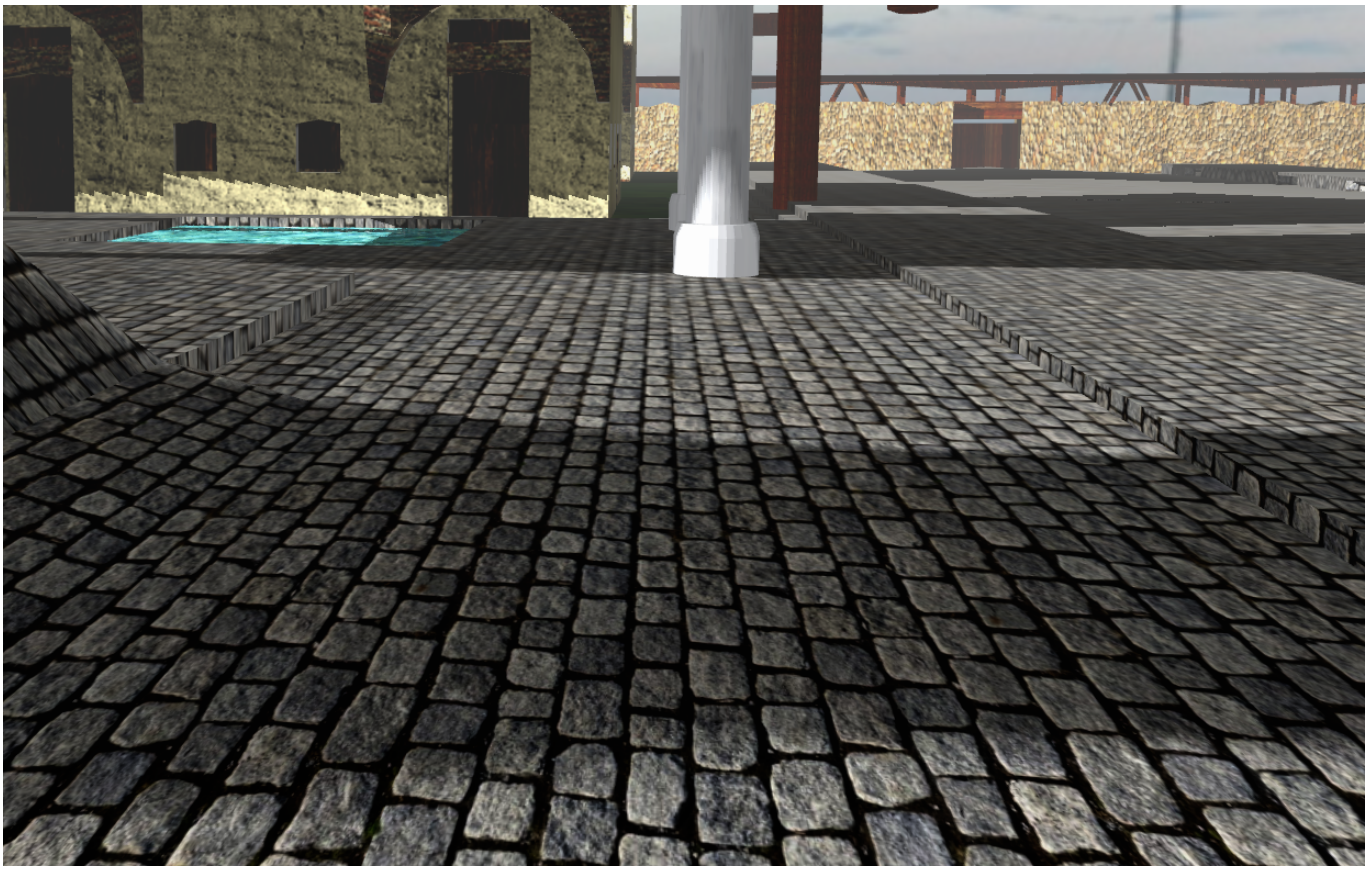
## Dynamic Percentage-closer filtering

It is used to create soft shadows or to create the apparent increase in shadow map resolution. In the project, it is implemented a special flavour of this algorithm, in the sense that it is calculated the distance from the fragment to the camera and depending on that, the resolution of the PCF will vary from a map of 1x1 (or a single pass) to a map resolution of 11x11 in order to create smooth and soft shadows when the player is close to an object. If the user is at a distance greater than 50 units, then the resolution will be 1x1. As the player gets closer, the resolution increases, until the camera is at 5 units, then the resolution remains at 11x11, in order not to burden the GPU with more calculations than it is necessary. This was implemented in order to simulate the multiple shadow map passes used in modern computer graphics.

```
float shadow = 0.0;
vec2 texelSize = 1.0/textureSize(shadowMap,0);
int rangeShadow = max(0, min(5,int(-0.111111*length(fPosEye) + 5.555556)));

for(int x=-rangeShadow;x<=rangeShadow;++x)
{
    for(int y=-rangeShadow;y<=rangeShadow;++y)
    {
        float pcfDepth =
texture(shadowMap,normalizedCoords.xy+vec2(x,y)*texelSize).r;
        shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
    }
}
```

A comparison between two PCF resolutions can be observed by looking at the shadow cast near the camera and on the water level:



## Water movement

Water level varies pseudorandomly using a LFSR random generator function[^6]. The function creates a random number with the seed representing the current time and fragment vertex position. Then, it gets added to a variable that assures that the water level flows in a pretty realistic way, without breakes in the average mesh of the object.

```
vec2 resolution = vec2(1024,1024);
vec2 cPos = -1.0 + 2.0 * vPosition.xy / resolution.xy;
float cLength = length(cPos);
vec2 uv = vPosition.xy/resolution.xy+(cPos/cLength)*cos(cLength*12.0-time*4.0) *
0.05;
vec3 newPosition = vPosition;
float disp = noise3f(newPosition*time);
newPosition.y = newPosition.y  + sin(disp + uv.y)/50.0f;
```

## Day-Night cycle with moving Sun and responsive street lights

This behavior is pretty simple in the sense that the directional light will change its vector to appear as if the sun is moving on the sky. What it makes it difficult, is the behavior of the directional light color and street lights that modify along with the sun's position on the sky.

Bellow it can be observed the position of the sun is calculated according to a variable of intensity [0,2] that is modified using the delta time variable, as well as, an addition to the current intensity.

```
if (intensity > 1.0f)
{
    newPosition = glm::vec3(-0.5f * 100 * (intensity-1) + 10, -1 * 100 *
(intensity-1) + 100, -3 * 100 * (intensity-1) + 10);
}
else
{
    intensity = 1 - intensity;
    newPosition = glm::vec3(0.5f * 100 * intensity + 10, -1 * 100 * intensity +
100, 3 * 100 * intensity + 10);
}
```

The street lights will change their color and intensity based on this `intensity` variable with the following meaning:

- (1.75f,2.0f] -> the in-game sunrise, the directional light will have the color `#5c2d04` and street light will have the 1.0f intensity (maximum)
- [1.5f,1.75f] -> the in-game morning, the directional light will have the color `#ffe4c9` and street light will have the

  ```
  pointLightsIntensity = 1.0f - (1.75f - intensity) * 4
  ```

- (0.5f, 1.5f) -> the in-game day, the directional light will have the color `ColorParser.WHITE` and street light will have the 0.0f intensity (minimum)
- [0.25F,0.5f] -> the in-game afternoon, the directional light will have the color `#ffe4c9` and street light will have the

  ```
  pointLightsIntensity = 1.0f - (intensity - 0.25f) * 4
  ```

- [0,0.25f)-> the in-game evening-early night, the directional light will have the color `#5c2d04` and street light will have the 1.0f intensity (maximum)

That can be concluded that the times of day that were implemented are symmetrical around the day part of the game, but they needed some alterations, especially when we talk about the formula of calculation

## Map Animation

The map animation has two important parts that work togheter, in order to provide the effect of map transition. This movement was inspired by various games that provide a map system to the player. These parts are:

- camera movement -> The algorithm stores the current camera position, then until the rotation on the x axis is greater than -90, it rotates downwards, moves the camera up on the y axis and renders the scene.
- camera projection type -> when the camera reaches pitch = -90, yaw = 90, then it switches from perspective to othographic projection.

Doing so, the camera has a smooth transition and the player is not entirely disorented by the animation. When the player wants to go back to the normal view, the algorithm works in reverse, in order to get the camera rotation and position to the exact same values as before.

```
while (rotation.x > -90) // when camera goes from street view to map view
{
    deltaTime.calculateDeltaTime(true);
    rotation = myCamera->getRotationAxis();
    myCamera->rotate(rotation.x - 5 * deltaTime.getRotationSpeed(), rotation.y);

    glm::vec3 cameraPosition = myCamera->getCameraPosition();
    myCamera->move(glm::vec3(cameraPosition.x, cameraPosition.y + 10 *
deltaTime.getTranslationSpeed(), cameraPosition.z));

    renderScene(myWindow, *myCamera, deltaTime);
    glfwPollEvents();
    glfwSwapBuffers(myWindow.getWindow());
}
```

## Music and positional sounds

The music interpreter is provided by the library irrklang[^7]. This is a high level 2D and 3D cross platform (Windows, macOS, Linux) sound engine and audio library which plays WAV, MP3, OGG, FLAC, MOD, XM, IT, S3M and more file formats, and is usable in C++ and all .NET languages (C#, F#, etc).

In the project, it is used to provide:

- atmospheric 2D music implemented in the ObjectManager.cpp file
- object 3D sound effect, in my case, the merchants in the stalls. The position of the sound is initialised in the InGameObject.cpp file and has the capability of moving with the object, this providing real feedback from the object.

Every frame, the sound engine has to recalculate the listener position (the camera position), in order to create true 3D feeling of the sound effects:

```
vec3df cameraPosition = vec3df(myCamera.getCameraPosition().x,
myCamera.getCameraPosition().y, myCamera.getCameraPosition().z);
vec3df cameraDirection = vec3df(myCamera.getCameraFrontDirection().x,
myCamera.getCameraFrontDirection().y, myCamera.getCameraFrontDirection().z);
SoundEngine->setListenerPosition(cameraPosition, cameraDirection);
```

## Graphics model

Almost every model was created by me in Blender using textures from internet^2^3. These sites only provided the ambient texture, so I've used the free tool NormalMap-Online^4 in order to create the Normal, Specular and sometimes the displacement maps. This part was very fun, because I really enjoyed creating something from scratch and seeing it in the world that I've created. The only model that was imported from internet^1 was the statue, because I do not posess such artistic skills.

For the sound effects, Freesound.org^5 and Pixabay.com^2 were the main sources, but I've also recorded my own voice for the merchants shouts.

## Data structures

An array of data structures were used. The most useful and interesting were:

1. POINT_LIGHT struct in .frag and Objectmanager.cpp

```
struct POINT_LIGHT
{
    vec3 location; // location of the light
    vec3 color;    // color of the light
    float intensity; // intensity of the light
    // 3 locations
};
```

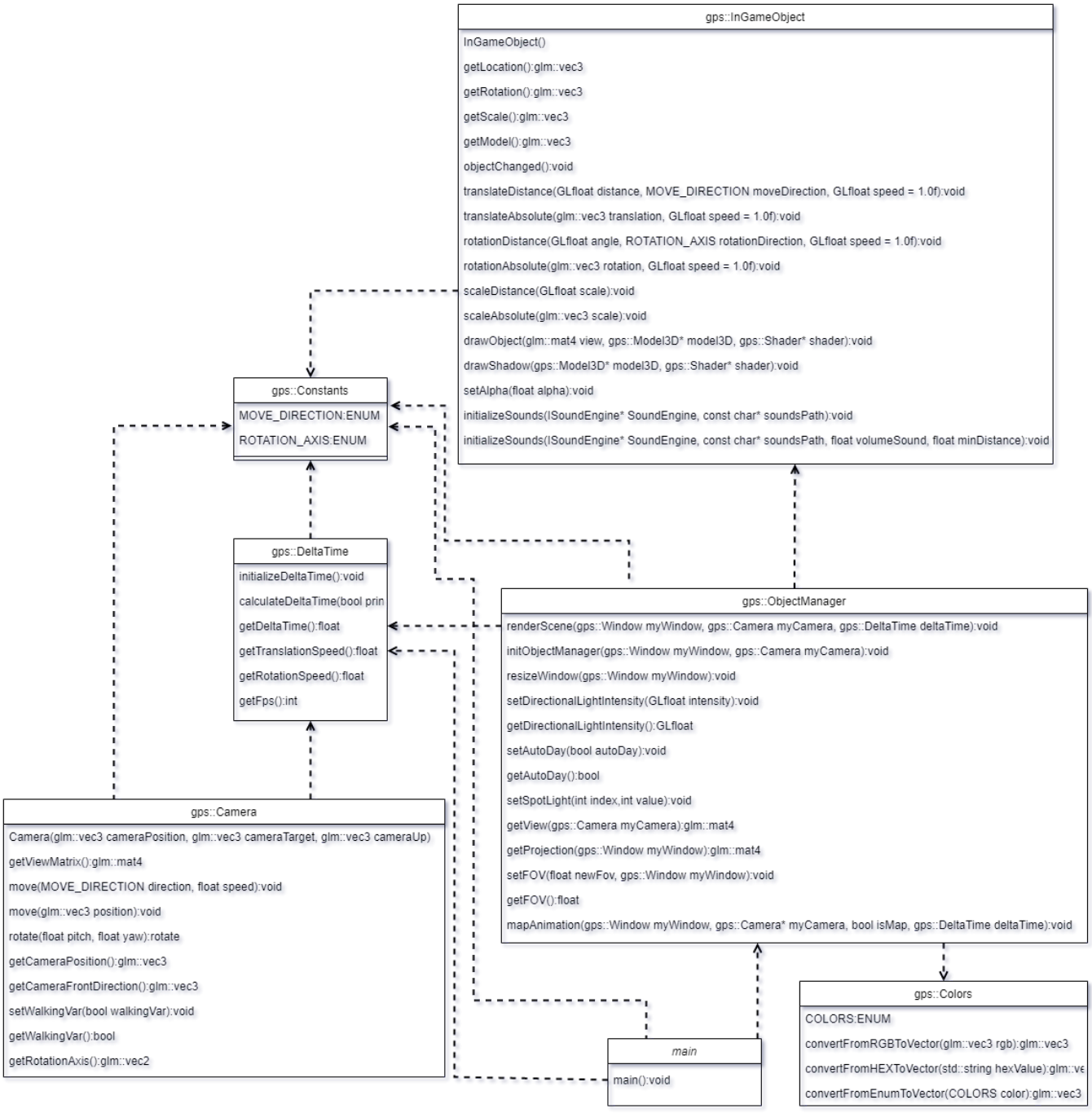
2. SPOT_LIGHT struct in .frag and Objectmanager.cpp

```
struct SPOT_LIGHT
{
    int spotinit; // if the spot light is on/off
    vec3 spotLightDirection; // vector for the light direction
    vec3 spotLightPosition; // location of the light
    // 3 locations
};
```

3. std::vectorgps::InGameObject object

This data structure was used in order to create enough instances of InGameObject classes to store the information of each object. It was chosen this behavior, in order to save resources when the application runs, as well as future proofing the implementation if a frustum culling or similar implementation is added later, which might require the programmer to dynamically control all the objects in the scene.

# Class hierarchy




The explanation of the UML is provided below:

1. Class gps::Constants is used to store and interpret the movement and rotation enums, in order to standardise these elements across the project
2. Class gps::Color was created to make the convertion between different color codes to gl color codes easier
3. Class gps::DeltaTime is used to store and compute the deltatime in the application, to have a more abstractised approach of this feature
4. Class gps::Camera is responsible for the camera related behaviour and computations
5. Class gps::InGameObject is used to store and compute the matrices and rendering of a single object
6. Class gps::ObjectManager was created as a game manager class, where all the features of the application can interract with each other in a controlable and abstractised manner

# Conclusions

This project was created as a trial of a simple, but effective library that can be used by anybody to create simple and basic prototypes in the OpenGL framework. It was an interesting experience that made me learn and research more on the OpenGL behavior and functionalities and it could be extended with the following:

- more abstractization overall
- more light sources types and modularity
- dynamic shadows from the light sources
- more atmospheric effects like rain, snow etc
- more sound effects capabilities
- animation of characters and objects

# References

1. Low Poly Roman Insula 1 by lexferreira89
2. Pixabay
3. Solar texture
4. NormalMap-Online
5. Freesound
6. Water waves pattern
7. irrklang