A Compact City Block

COMP 371 - Computer Graphics

Computer Graphics Project Report

Submitted to Kaustubha Mendhurwar & Sudhir Mudur

By

Anik Patel (40091908)

Ian Lopez (27296126)

Gregory Alexis John (40041536)

Jonathan Andrei (40051683)

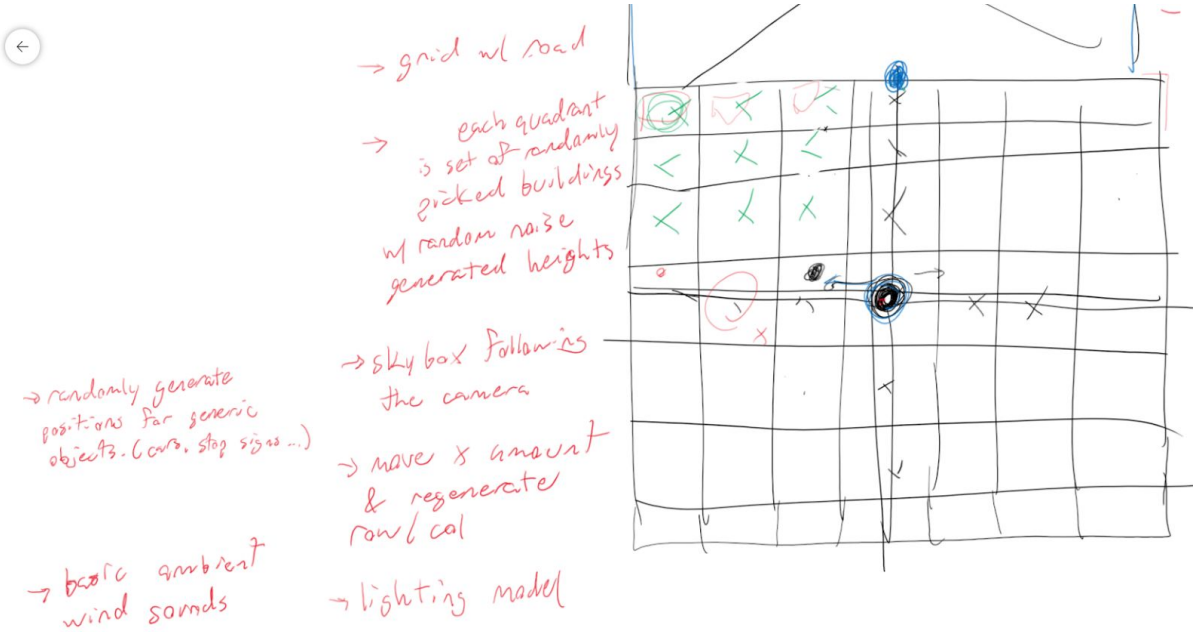
April 19th, 2020

1.0 Introduction

This project had three directions that we could have taken: creating an interactive museum, creating a procedurally generated world using models, or by creating an animated narrative story using OpenGL.

1.1 Objective

We decided to take on the task of creating a procedurally generated world, by creating a city landscape using some free 3D models that we found online through various websites such as turbosquid and free3d. We first got together and made this plan:



Essentially creating a city by splitting it into a grid where each cell is qualified as either a road, where we could generate cars, light posts, bus stops, etc., or a building, where we would select one of our building models to be rendered with a randomized height so that there would be some variation in the building that would be rendered. We had a lot of ideas and objectives planned out for this project, but not all ended up doable.

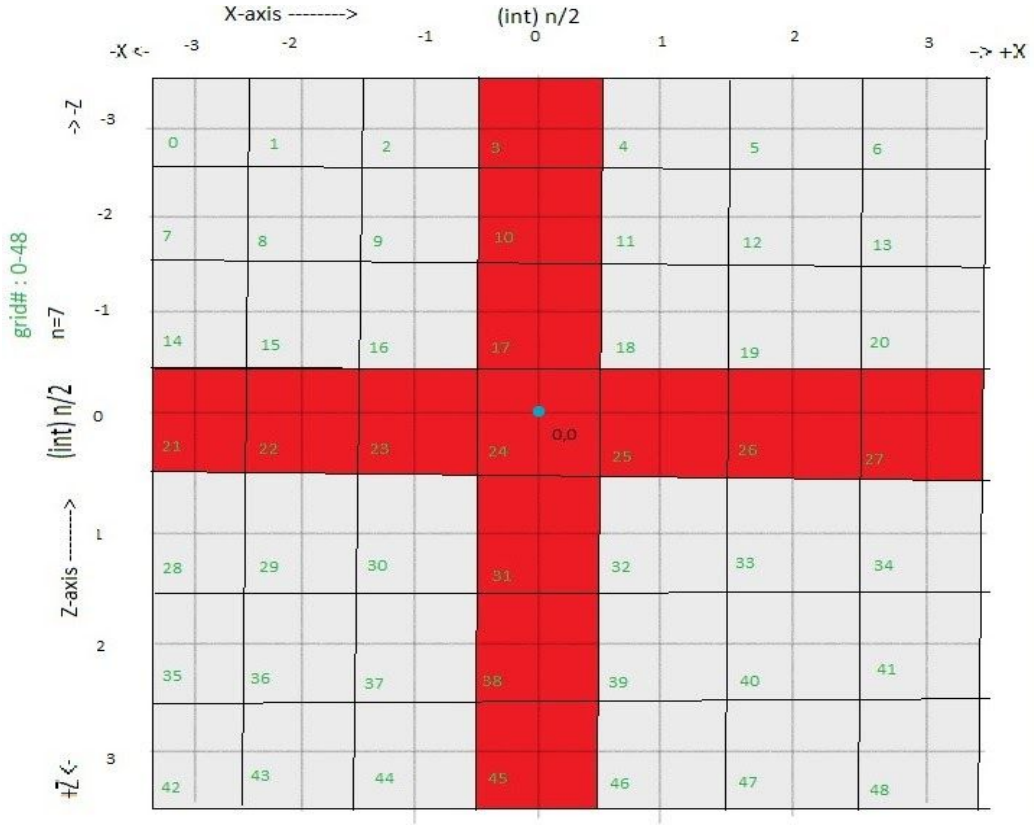
1.2 Initial Timeline

When the group project was first announced, our group was ambitious to get some sort of procedural generated world. Although there was discussion amongst the group to create a jungle or some sort of western town, we ultimately decided on an urban city, with roads, buildings, and all types of metropolitan aspects. We went through all the steps imaginable to make this project come to life. We have set up a trello (a website used to track group projects and individual tasks) and more importantly we set up a github repository to make our lives easier. Finally deciding on what type of project we were going to make, we split the tasks up amongst the four and went to work with weekly deadlines and weekly roundups talks on what to work on next. One of the biggest challenges we faced, as well as every other student across Concordia, was the COVID-19 outbreak. There was a time when the group was a bit in limbo on whether to proceed on everything we had originally planned for our world. Fortunately, most of our original ideas came into fruition as you will see when our code and final product is reviewed.

2.0 Project Implementation

2.1 Grid System

The grid system is used to compartmentalize the different objects spawned and makes it easier to manage the different varieties of objects and where they are spawned, such as roads, buildings, cars, lamp posts etc. We decided to create a n*n sized grid, where n is any positive integer, and each square inside the grid is 1*1 unit^2 in area. We decided that the centre lines of X axis and Z axis will be squares reserved for roads and the rest of the squares will be where other objects such as buildings will be spawned. To implement this system, we made a Grid.h and Grid.cpp files, which will generate the necessary number of squares based on the given n, and these squares will be stored in 2 <vector>s, for the X-axis and Z-axis coordinates. We're storing the centre point of each square in these. Given below is a visual representation for a 7x7 grid. We have implemented 2 functions that return a



vector of pairs of X and Z coordinates, containing spawn points for the roads and buildings. As they return the centre points of each square, this can be used to spawn models in the middle of the square and can be scaled up or down as per the need be. Other objects such as trash cans, lamp posts and vehicles are spawned relative to the centre coordinates of the road squares. The grid system is designed in such a way that no matter what the value of n is, the centre point of the central block of the grid will always be at the (0,0) coordinates and the roads will always lie on the X-axis and Z-axis.

2.2 Generic Item Generation

The generic item generation portion applied here was a bit of the challenge to figure out. What I wanted to achieve is the most realistic set up for an urban street.

```
std::random_device rd;  //Will be used to obtain a seed for the random number engine
std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()
std::uniform_real_distribution<> dis(-45.0, 45.0);


//cars
objMesh lexus("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(dis(gen), -0.05f, -1.0), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus2("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(-10.0f / 12, -0.05f, dis(gen)), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus3("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(dis(gen), -0.05f, 1.0), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus4("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(10.0f / 12, -0.05f, dis(gen)), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus5("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(dis(gen), -0.05f, -1.0), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus6("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(-10.0f / 12, -0.05f, dis(gen)), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus7("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(dis(gen), -0.05f, 1.0), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
objMesh lexus8("assets/models/lexus.ob", glm::vec3(.99f), glm::vec3(10.0f / 12, -0.05f, dis(gen)), glm::vec3(0.4f / 12, 0.4f / 12, 0.4f / 12));
std::uniform_real_distribution<> dis2(0.0, 1.0);
lexus.setColor(glm::vec3(dis2(gen), dis2(gen), dis2(gen)));
lexus2.setColor(glm::vec3(dis2(gen), dis2(gen), dis2(gen)));
```

To mainly generate the position on the x and z axis position of all cars. This function was also implemented to the colors. You will note that each car will have a different color and position per map generation. Other than that I created a set of light poles, bus stop benches, and trash cans in a grid like fashion using a for loop. If we notice the image above we will see that a generated number will be placed inside the vec3 for a lexus' x or z coordinate (as well as a color for random color).

2.3 Building Rendering

From the models that we found online, we picked simplistic low polygon count models. We were inspired by how the buildings look in the background of the main menu for the game Mirror's Edge, made by Dice. We liked how they made non-detailed models of the buildings look.

Screenshot from Mirror's Edge Main Menu

By having the models that we use be simple and not too detailed, we can make sure that we are able to run our project on lower powered computers such as some of our laptops. This can also ensure lower load times since there are not too many vertices to load.

Once we had our buildings selected, we want to add variations in how they looked, since if by chance we get the same building picked back-to-back, we would end up with a scene with the same models and the exact same heights. So we used the C++ pseudo-random number with a random seed to pick the heights.

```
auto buildings = gr->getBuildingPos();
for (auto i : buildings)
{
    srand(seed);
    int buildingChoice = rand() % 6;
    float size = randomizeHeight(.05f, .2f);

    modelsList.push_back(new objMesh(buildingOptions[buildingChoice], glm::vec3(1.f, 1.f, 1.f), glm::vec3(i.first * 10.f, 0.f, i.second * 10.f), glm::vec3(.05f, size, .05f)));
    seed += rand();
}
```

Code for Randomness for Buildings

2.4 Skybox

The skybox used are six images of a volcano, strategically labelled by the side of the cube.

[6] *Skybox Images*

Each image is loaded into a vector called faces and positioned at the origin. The texture coordinate will access each face to give the illusion of a joint landscape. A function in the Texture class takes in the faces vector as a parameter and creates a texture ID, binding to GL_TEXTURE_CUBE_MAP. In addition, a wrap and an edge is set in 3-dimensional space to convey the environment. Moreover, a loop is created to set the image at each axis, starting at the enum coordinate GL_TEXTURE_CUBE_MAP_POSITIVE_X incrementally to GL_TEXTURE_CUBE_MAP_NEGATIVE_Z.

```
unsigned int loadCubemap(std::vector<std::string> faces)
{
    unsigned int textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_CUBE_MAP, textureID);

    int width, height, nrChannels;
    for (unsigned int i = 0; i < faces.size(); i++)
    {
        unsigned char* data = stbi_load(faces[i].c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,
                0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data
            );
            stbi_image_free(data);
        }
        else
        {
            std::cout << "Cubemap tex failed to load at path: " << faces[i] << std::endl;
            stbi_image_free(data);
        }
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);

    return textureID;
}
```

In addition, within the main.cpp file, an array of the skyboxVertices is created and a separate VAO is reserved for the skybox. The skybox images are initiated within a vector, which are used in the final render.

```
// Skybox Textures
std::vector<std::string> faces{
    "assets/textures/skybox_images/right.png",
    "assets/textures/skybox_images/left.png",
    "assets/textures/skybox_images/top.png",
    "assets/textures/skybox_images/down.png",
    "assets/textures/skybox_images/back.png",
    "assets/textures/skybox_images/front.png"
};
```

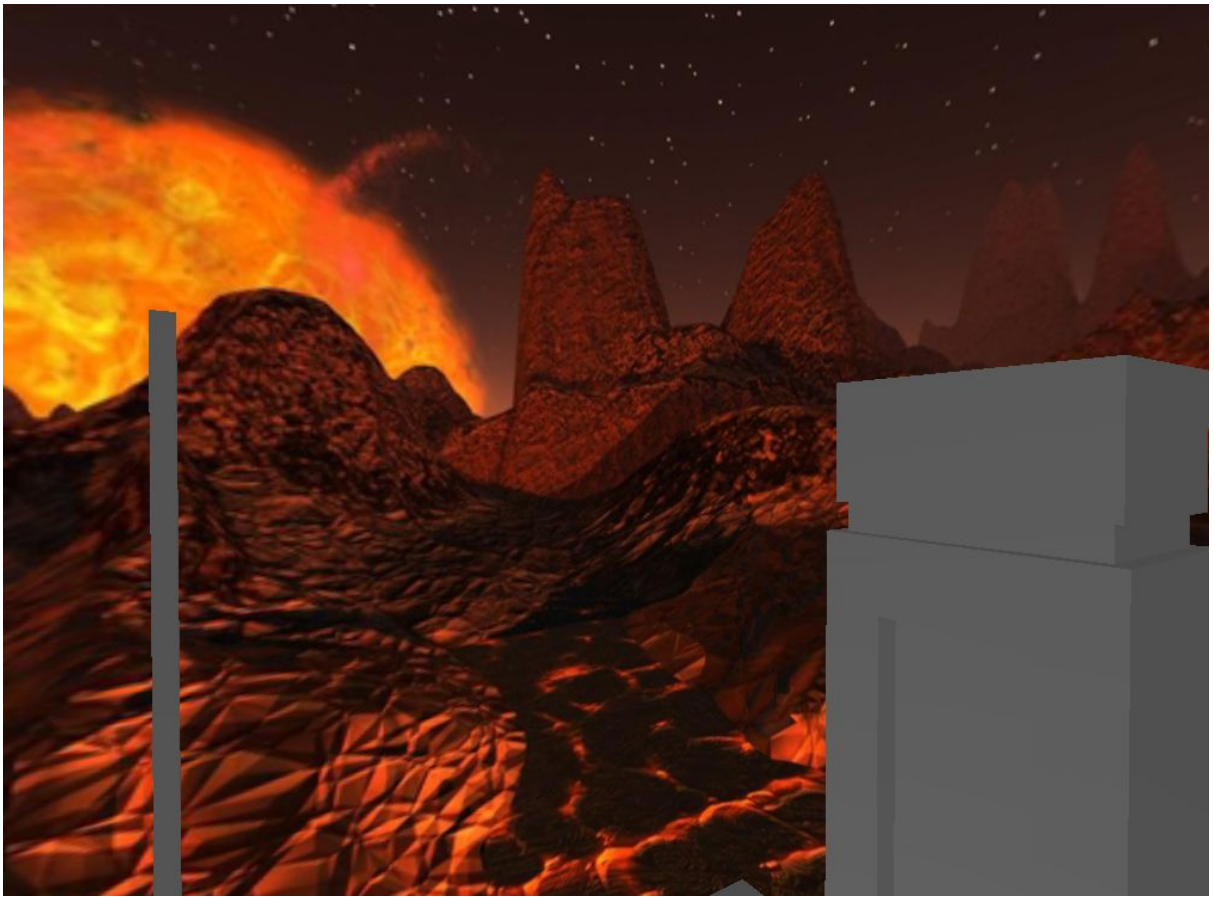
Furthermore, within the shaders folder under assets, a separate vertex and fragment shader is created dedicated to the skybox. In the vertex shader, the z-value is set to w, making it have the unit value 1. This implementation allows the skybox to output wherever no models are in the camera's point of view.

```
#version 330 core
layout (location = 0) in vec3 aPos;

out vec3 TexCoords;

uniform mat4 projection;
uniform mat4 view;

void main()
{
    TexCoords = aPos;
    vec4 pos = projection * view * vec4(aPos, 1.0);
    gl_Position = pos.xyww;
}
```

Vertex Shader

Furthermore, when the skybox is drawn in our loop, it is rendered last, to render all the models and other elements first. This way, resources are used in a more optimal way to

render only the images of the skybox that are detected to have no other models in front of it. As a result, a glimpse of the skybox and how it distinguishes itself from the model is below:

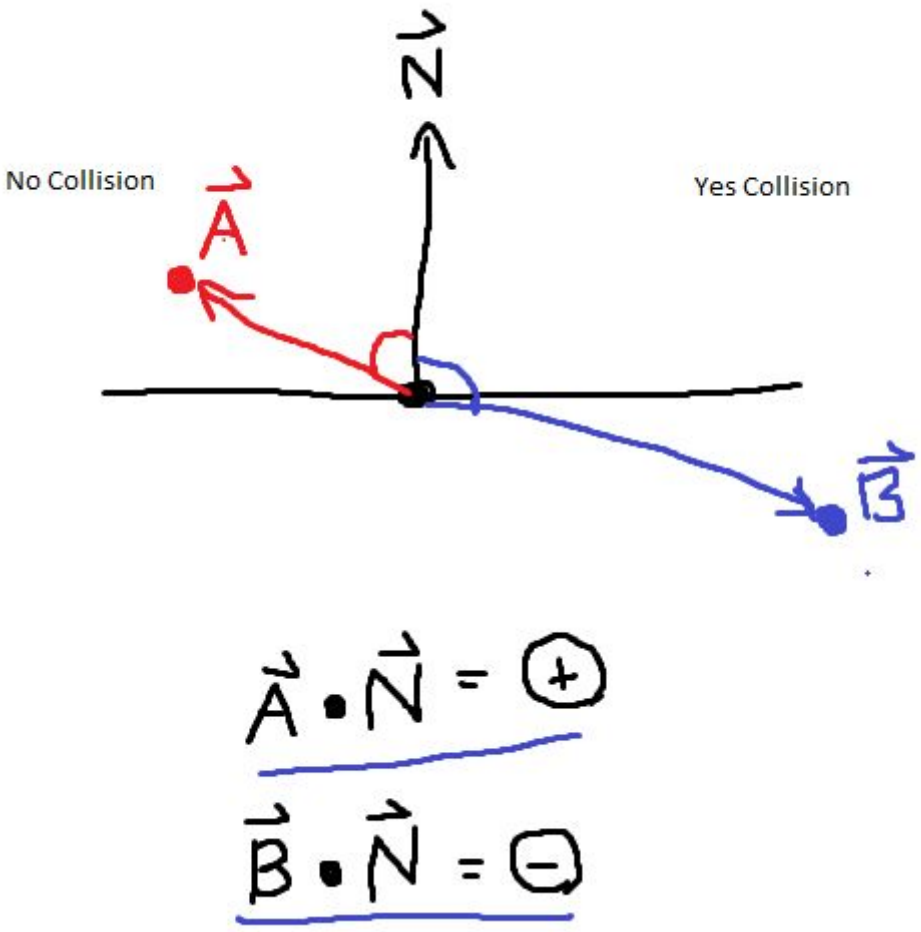


2.5 Collision Detection for Walls

The collision detection that we implemented is very simplistic, essentially creating a few invisible walls that the camera can not go through. We accomplished this by checking the result of a dot product between a vector that is from a point on the wall itself to the camera's current position and the normal for that wall. We compare that result to a range which tells the camera how close it can get to the invisible wall before getting stopped.

```
bool Camera::checkCollision(glm::vec3 pos, glm::vec3 norm, float range)
{

    return glm::dot(norm, *camEye - pos) < range;
}
```

Code Snippet, similar to code from lab



Math used for Collision Detection

If a collision is detected, we would reset the position of the camera in a way so that it stays on the walls and never goes beyond the limitations of said wall. In our current project implementation, there is a wall for the ground plane and for each side of the roads, so the camera is not able to go inside, or even that close, to the buildings that are rendered.

3.0 Conclusion

We learned quite a few things throughout working on this project. First off, we had to learn to adapt and work with the situation that occured with the pandemic that started in the middle of the school semester. Especially with the full week that was taken away from school, every class had to put all their assignments and projects due to be submitted at the same time, but we managed to organize our time in a way to accomplish our goals. We also learned quite a few things that were not covered during the lectures, or that might have been missed by us not being able to attend all the labs, such as using a cubemap for the skybox and how to perform collision detection.

There are many aspects that could be improved. The first and most obvious thing would be the way we have our models loaded. We used a combination of object-oriented programming with the model loading function that was provided to us during the labs. In our case, the initial loading time of the program is very long since it loads the model for each variable we have, meaning it loads the same models over and over again each time if another instance of the model is used. A way we could improve on this would be to separate the loading of the model file and the actual model being rendered. That way we can pass the VAOs and VBOs into the model so we only need to load each file once. On this same train of thought, we should work on getting the code properly adhering to regular C++ programming practices such as actually managing the memory and separating parts out of the main file such as the shadow mapping and skybox creation. All in all, we were able to accomplish work that we can be proud of.

4.0 References & Resources

[1] DHK_krm.'bus stop plastic', 2016 [Online]. Available: https://www.turbosquid.com/FullPreview/Index.cfm/ID/1056405 [Accessed: 10 - Apr - 2020].

[2] tyrosmith. 'Street Light (Lamp)', 2015 Available: https://free3d.com/3d-model/street-light-lamp-61903.html [Accessed: 10 - Apr - 2020].

[3] tyrosmith. 'Stop Sign', 2013 Available: https://free3d.com/3d-model/stop-sign-58062.html [Accessed: 10 - Apr - 2020].

[4] stuart.'Lexus',2013, Available: https://free3d.com/3d-model/lexus-11015.html [Accessed: 10 - Apr - 2020].

[5] backlog_s, '19 Low Poly Buildings', 2018, [Online]. Available: https://free3d.com/3d-model/19-low-poly-buildings-974347.html. [Accessed: 10 - Apr - 2020].

[6] Elyvisions, 'Elyvisions Skyboxes', 2015, [Online]. Available: https://opengameart.org/content/elyvisions-skyboxes . [Accessed: 13 - Apr - 2020]

[7] Joey de Vried, 'LearnOpenGL - Cubemaps', 2014. [Online]. Available: https://learnopengl.com/Advanced-OpenGL/Cubemaps . [Accessed: 16 - Apr - 2020]