TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING
IOE CENTRAL CAMPUS, PULCHOWK

COMPUTER GRAPHICS
Project Documentation

on

CONTROL SYSTEM BASED SELF DRIVING VEHICLE SIMULATOR

Submitted By:
Sakar Pathak (075BEI030)
Shambhavi Adhikari (075BEI034)
Shreejan Singh Silwal(075BEI035)

Submitted To:
Department of Electronics and Computer Engineering
December 6, 2020

1. Acknowledgement

We would like to express special thanks of gratitude to our teacher Mr. Anil Verma and Mr. Suresh sir as well as our department Electronics and Computer Engineering for giving us the golden opportunity to do this wonderful project on the topic Control system based self-driving vehicle simulator, which also helped us in doing a lot of Research and we came to know about so many new things we are really thankful to them. Secondly, we would also like to thank our parents and friends who helped us a lot in finalizing this project within the limited time frame. We would like to express our deepest appreciation to all those who provided us the possibility to complete this report. Last but not least, we have to appreciate the guidance given by other supervisor as well as the panels especially in our project presentation that has improved our presentation skills thanks to their comment and advices.

2. Abstract

An autonomous vehicle driving control system carries a large number of benefits, whether for the engineering industry or engineering education. The system discussed here provides the measurements obtained from vision-namely, offset from the centerline at some look-ahead distance and the angle between the road tangent and the orientation of the vehicle at some look-ahead distance-and these are directly used for control. This paper also presents simulation results regarding autonomous vehicle dynamics and control, along with methods, techniques and approaches for developing this system which can be used by engineering educators for automotive, robotics or image processing courses.

3. Table of contents

4. Introduction

Automation of the driving task has been the subject of much research recently, but to design and simulate an autonomous vehicle driving control system is not easy, because it is such a complex system. Over the last decades, many engineers and educators had problems in estimating the results of an autonomous vehicle driving control system, but, in recent years, computers have been used to great effect in simulating such systems. Therefore, the main objective of the system discussed in this paper is to demonstrate an autonomous vehicle driving control system to engineers, students and researchers by means of a simulated system. Another objective of this system is to prove that the steering command for the vehicle lateral control can be determined by processing and analyzing images taken whilst driving a vehicle. Then, by combining the steering command and other vehicle dynamics parameters, the vehicle's dynamics performance can be determined too.

In the real world, autonomous vehicles face complex and varied external environments. A good simulator decomposes external environment into the basic elements, and then rearranges the combination to generate a variety of test cases, each simulating a specific scenario.

As we know, finding the best control system design for the self-driving vehicle physically takes a lot of time and resources. So, instead of physically performing the tests and experiments, using simulation environment can save a lot of time, money and human resource.

There are two variables in our car:

1. Speed
2. Steering i.e front tire angle

5. Problem Statement

This control system driven self-driving car simulator is the simulation of real-life self-driving car. While making a self-driving car, if it is physically tested and experimented then it can cost a lot in terms of time and resources. So, with the self-driving car simulator we can simulate, the car giving any control system techniques like, PID control, LQR control, State Space control, etc. The results are equivalent to the real-life performance.

6. Objective

This study is aiming on producing a simulation of self-driving car based on control system that can self-drive by giving any control system techniques like PID control, LQR control, State space control etc. The objectives of the study is mainly:

i. To build a simulation capable of representing self-driving car.

ii. To study the OpenGl and implement it in our project.

iii. To implement the different computer graphics algorithm.

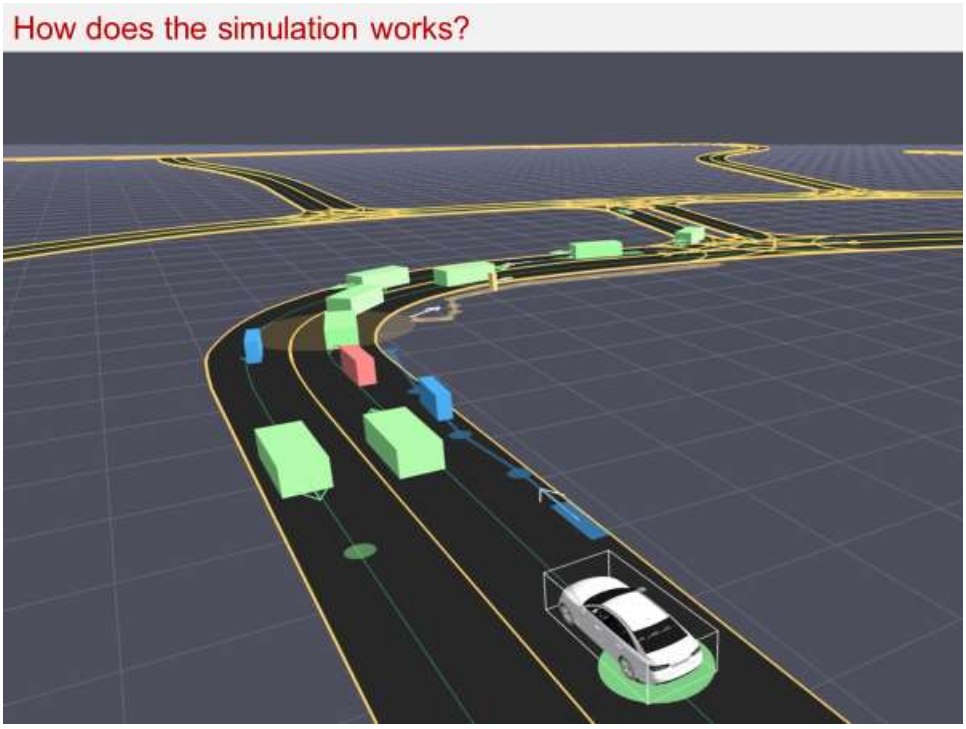
The scope of study for this project is: Software: Studying the algorithm used in self-driving car and developing a hybrid algorithm that will perform better in the real environment.

7. Working Method

The first part of the project is the computer graphics section. We have made a low-poly environment with blender and the environment is loaded with Assimp in OpenGL and rendered in screen. After that numbers of low poly buses, cars, vans etc. are made in blender and again loaded and rendered in screen. For lighting phong lighting model is used to make everything realistic and nice. For light a low poly sphere is placed above the environment and given the color of light to make it look like sun. This whole process requires lots of mathematics and matrix operations which are used for lighting, for scaling, reflection, translation and other transformations of models.

The second part of the project is the control system part. The self driving car can be loaded in the environment and simulated. It looks for the obstacle in its way and if the obstacle is found it changes its path by changing the direction and speed. It can take many control system techniques like bang-bang control, proportional control, PID control, LQR control, state space control, etc.

1. How does the simulation work?



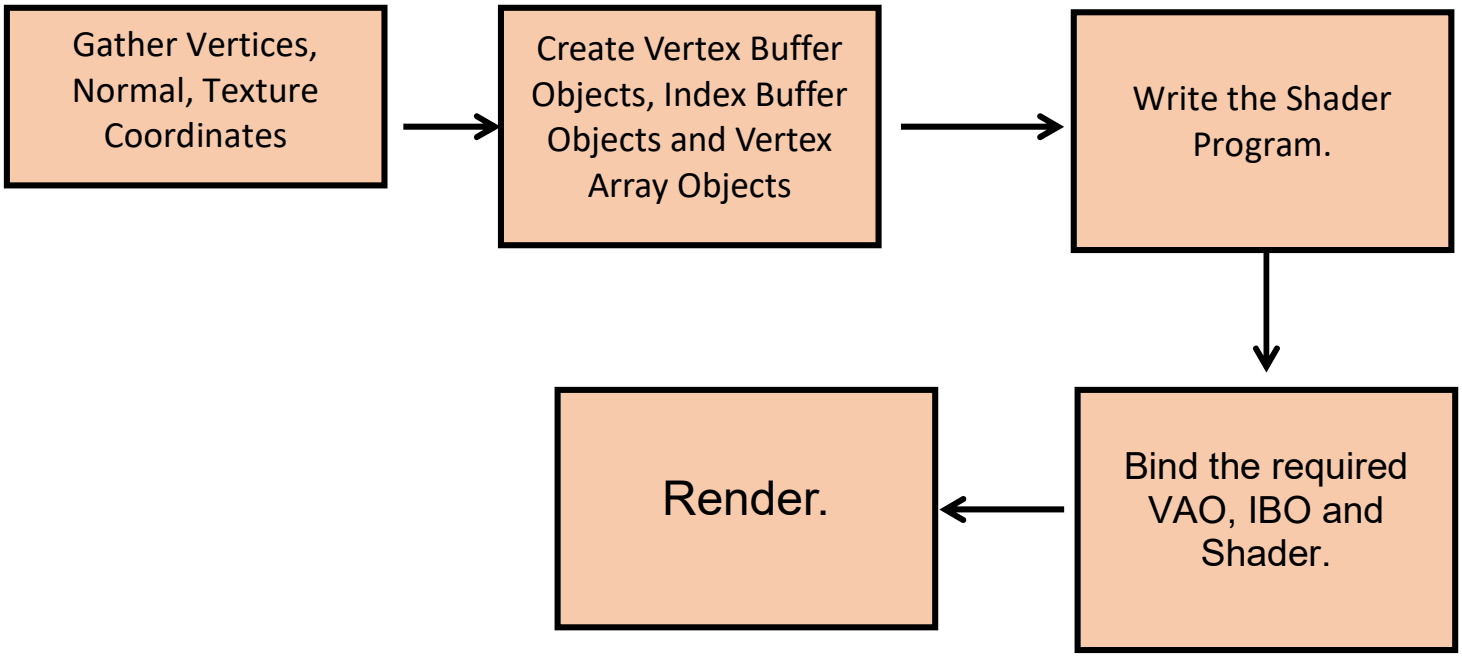
The vehicles roads and every model in the environment has their own boundary and if any other model touches the boundary of the other model then collision happen. So, the criteria of the self-driving car is to move on its path with such direction and velocity so that collision does not happen.

The self-driving Car has two variables:

1.Speed: the instantaneous speed of the self driving car changes after it sees and tries to avoid the obstacle. The change depends upon the control system used by us.

2.Direction: the instantaneous direction of car also depends upon the control system used after it sees the obstacles.

How OpenGL is used?



At first vertices, normal, texture coordinates are gathered and then vertex buffer objects, index buffer objects, vertex array objects are made. Shader program is written. Then required Vertex array object, index buffer object and shader is binded. After that the OpenGL can render the model in the screen.

What we've used?

GLFW: To create context window.

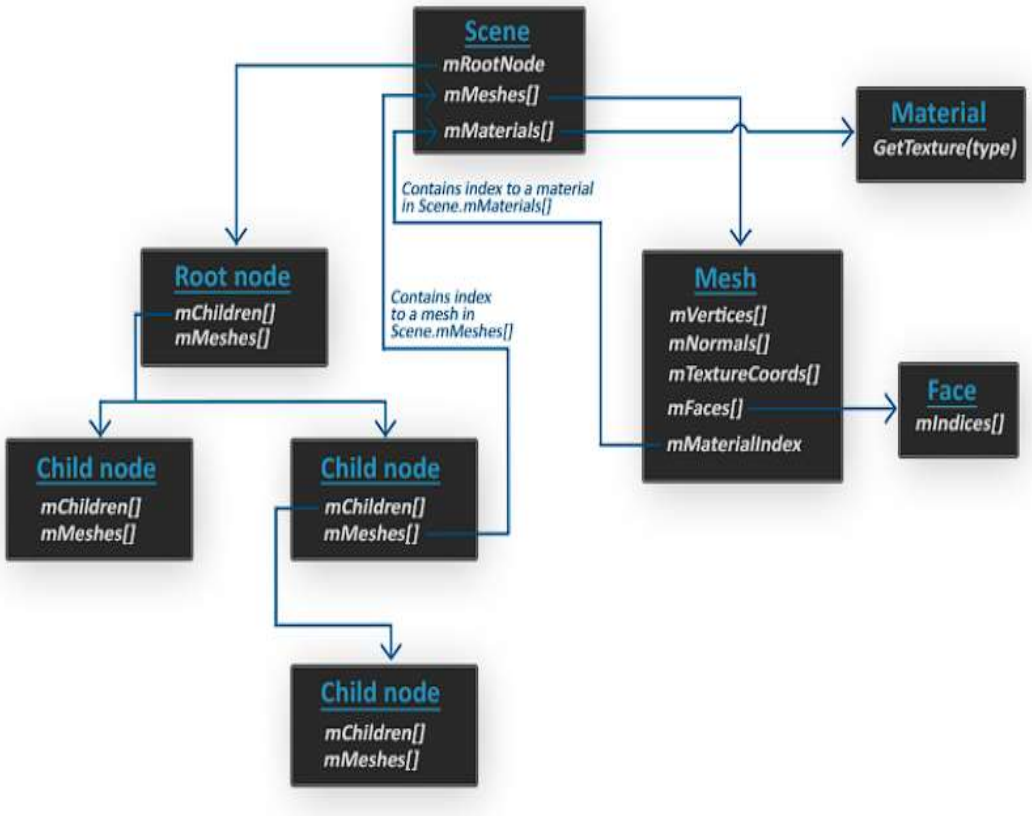
GLEW: To use Modern OpenGL Functions specific to the running device.

GLM(OpenGL Mathematics): for mathematical operations.

ASSIMP(Open Asset Importer): To import .obj files.

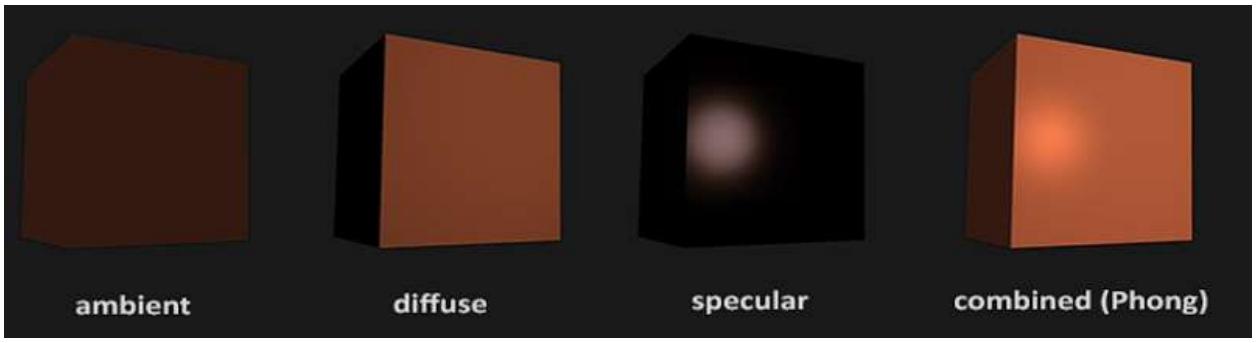
Loading and Rendering Models:

At first, required model is made using 3d modeling software like blender, fusion 360, maya, etc. Then, the model is exoported in .obj format which also exports a .mtl file linked with .obj. After that Assimp is used to read the .obj file which not only reads the .obj but reads the material and textures for us making our work a lot easier. Each model is made up of number of meshes. Therefore Assimp takes care of that and maintains a child parent relationship between meshes.



Lighting:

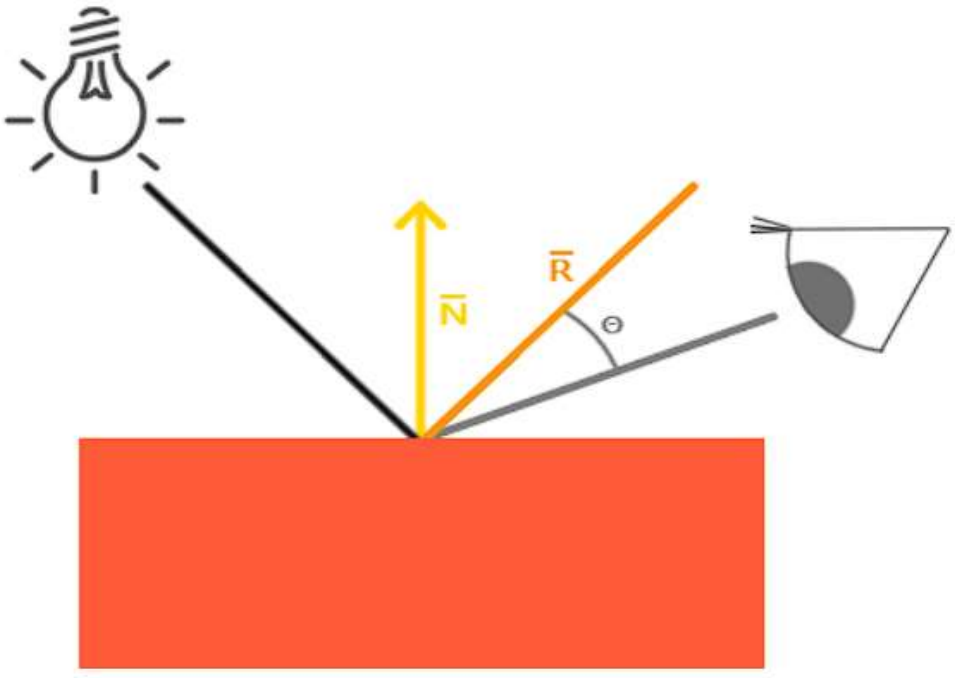
For lighting Phong Lighting model is used. The phong lighting model consists of ambient, diffuse, and specular lighting.



Ambient: Ambient light is the light present in the environment due to the reflections and many other light sources and noises.

Diffuse: Diffuse light is the actual light/color reflected by the material.

Specular: Specular is the shininess property of the material.



The light reflected from a fragment to the camera/eye depends upon:

1.Angle between the normal of the fragment and the light source.

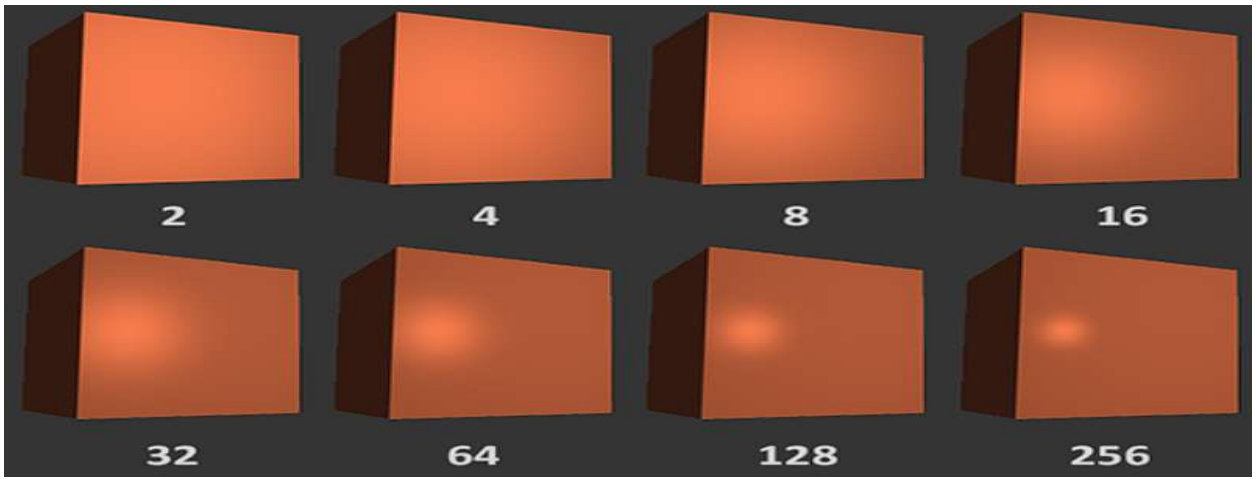
2.Angle between the eye line with the reflected light ray.

The use of ambient diffuse and specular light can be seen below.

```
vec3 ambientLight = material.ambient * light.color * light.ambient;

vec3 norm = normalize(Normal);
vec3 lightDir = normalize(light.position - FragPos);
float diff = max(dot(norm, lightDir), 0.0);
vec3 diffuseLight = (diff + 0.8) *material.diffuse* light.color* light.diffuse;

vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0), material.shininess);
vec3 specularLight = spec * material.specular * light.color * light.specular;

color = vec4((ambientLight + diffuseLight + specularLight), 1.0f);
```

Shininess of material/ mesh:

The shininess of the mesh shows its reflection property as shown below.



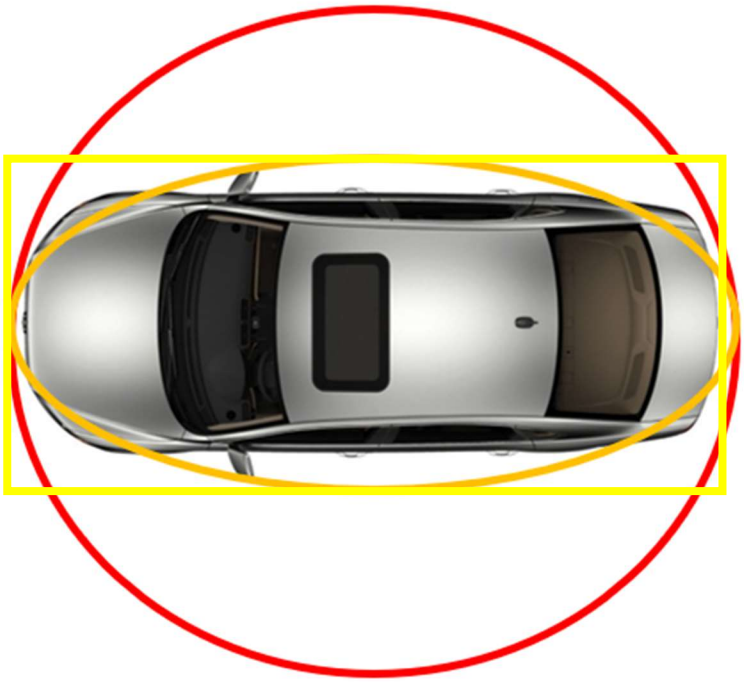
Collision:

For collision we have used circular and ellipse and rectangular boundary.

The circular boundary technique is the easiest collision technique because we have to keep track of only the center and radius of the circle. Below, the car is circled by the circle. So, if any other object tries to cross the circular boundary it collides.

And, in case of ellipsoid boundary we have to keep track of the center of ellipse and length of major and minor axis of the ellipse and also the yaw pitch and roll of the ellipse.

And in case of rectangular boundary we have to keep track of length and the breadth of the rectangle along with the center of the rectangle and also the yaw pitch and roll of the ellipse.



8. Algorithm

The first part of the project is the visualization, the visualization is done using OpenGL.

For visualization:

Step 1: For visualization OpenGL is used. We have designed environment and car models in blender and exported.obj files.

Step 2: The files are read using Assimp library along with materials and textures.

Step 3: Phong Shading is used in Shader program.

Step 4: Collision is used using circular, ellipse and rectangular boundaries.

Step 5: Camera angle can be toggled to random view and focused view. In random view you can change the camera direction and position using mouse and keyboard while in focused view the camera follows the car.

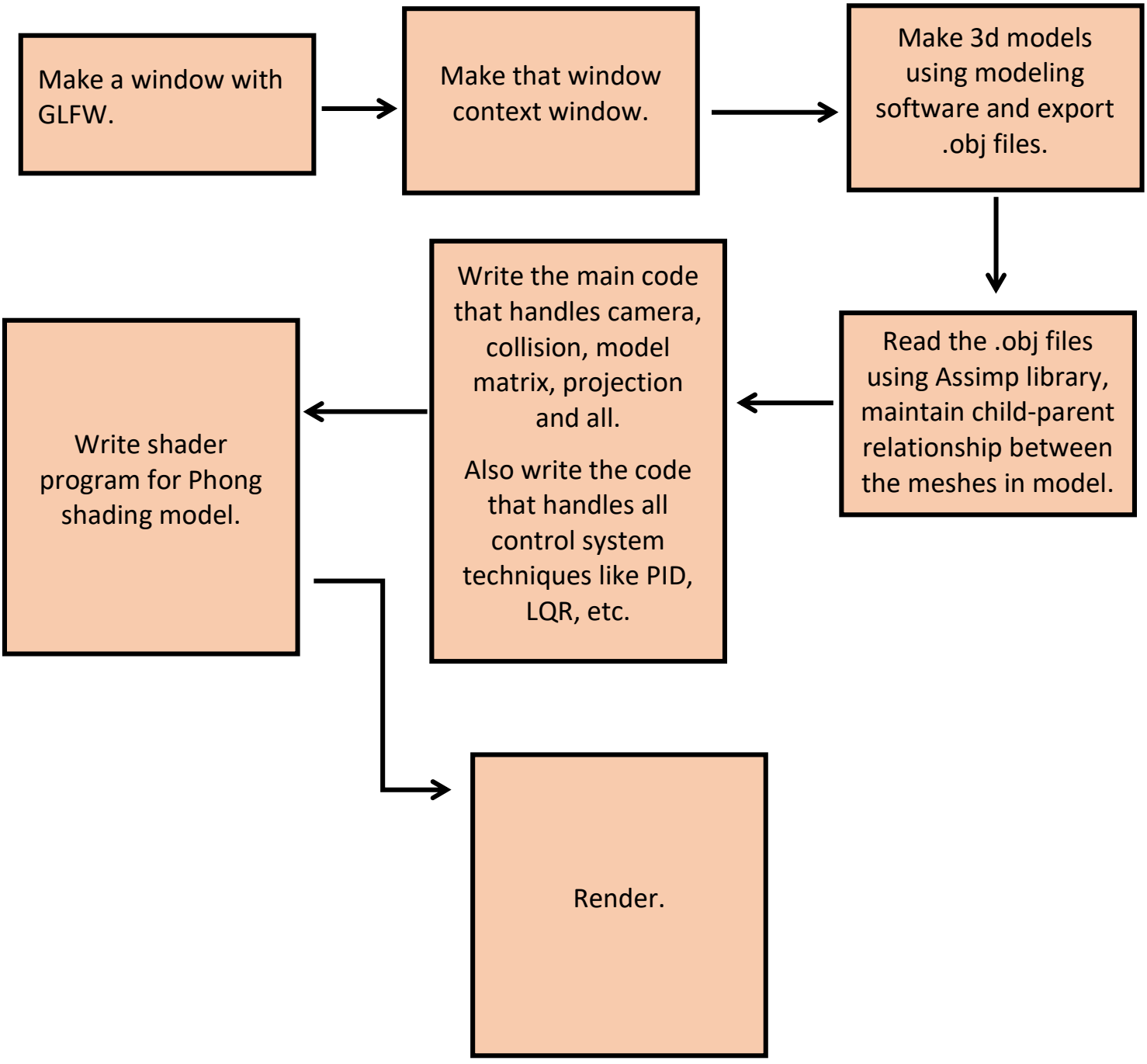
Step 4: The models are rendered in screen.

For Simulation:

The simulation part is actually a control system section instead of Computer graphics. Here different control system techniques like PID, LQR, state space control, etc can be implemented and the result can be visually seen and verified.

9. Flowchart

For visualization



10. Conclusion

Here in this project, we were able to magnify our ability to do C++ programming. With the end of this project, we were able to learn about different computer graphics algorithm. This project is very helpful for learning OpenGL and other different cross language application. We also could get overview of control system used in self-driving car.

11. References

https://www.geeksforgeeks.org/

https://learnopengl.com/

http://docs.gl/

https://www.khronos.org/

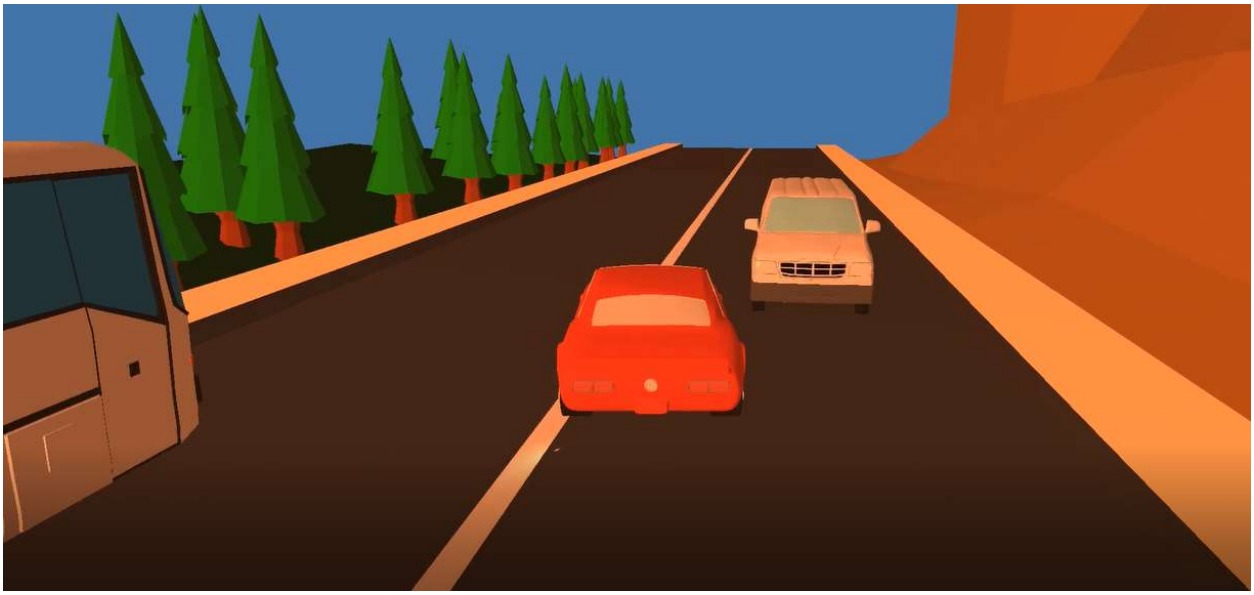
https://www.wikipedia.org/

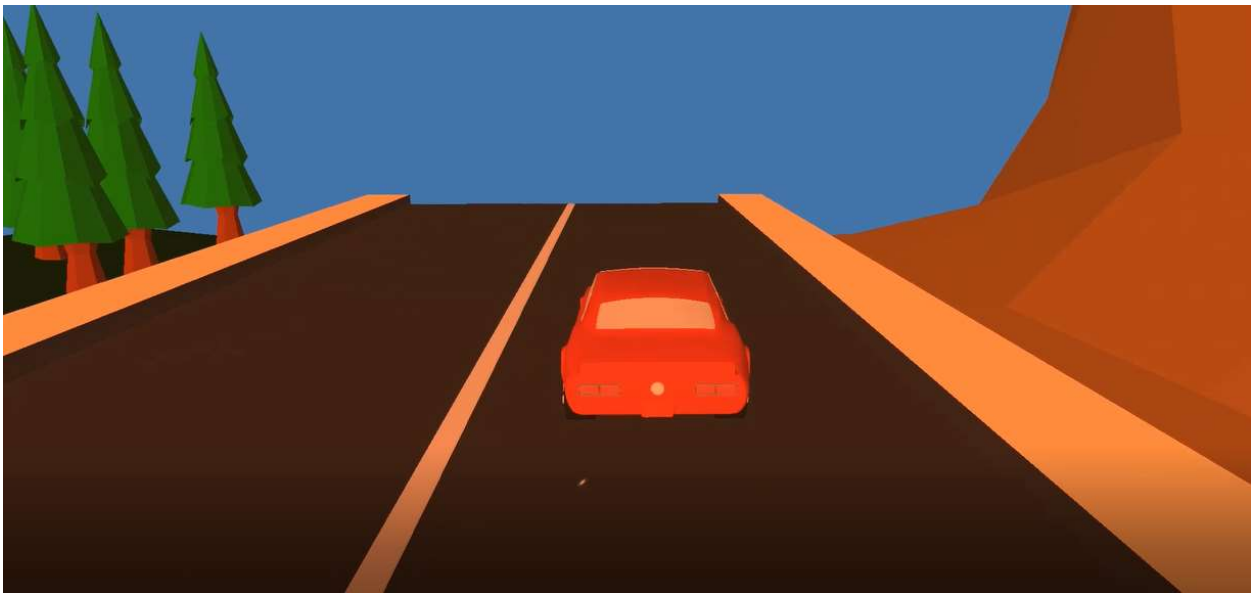
https://www.researchgate.net/

https://www.hurna.io/

12. Appendix

Image of self-driving car avoiding the obstacle:





GLSL CODE:

LightingSource.shader

```
#shader vertex
#version 330 core

layout(location = 0) in vec3 position;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
	gl_Position = projection * view * model * vec4(position, 1.0);
};



#shader fragment
#version 330 core

out vec4 color;
uniform vec3 lightColor;

void main()
{
	color = vec4(lightColor, 1.0);
    };
```

Model.shader

```
#shader vertex
#version 330 core

layout(location = 0) in vec3 position;
layout(location = 1) in vec3 normal;

out vec3 FragPos;
out vec3 Normal;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform mat3 transpose_inverse_model;

void main()
{
	gl_Position = projection * view * model * vec4(position, 1.0);
	FragPos = vec3(model * vec4(position, 1.0));
```

```
	//Normal = normal;
	//Normal = mat3(transpose(inverse(model))) * normal;    //matrix inversion in
shaders are costly since they are evaluated many times in a frame
	Normal = transpose_inverse_model * normal;
};



#shader fragment
#version 330 core

in vec3 FragPos;
in vec3 Normal;

out vec4 color;


uniform vec3 viewPos;

struct Light
{
	vec3 color;
	vec3 position;

	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
};

struct Material
{
	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
	float shininess;
};

uniform Light light;
uniform Material material;

void main()
{
	vec3 ambientLight = material.ambient * light.color * light.ambient;

	vec3 norm = normalize(Normal);
	vec3 lightDir = normalize(light.position - FragPos);
	float diff = max(dot(norm, lightDir), 0.0);
	vec3 diffuseLight = (diff + 0.8) *material.diffuse* light.color* light.diffuse;

	vec3 viewDir = normalize(viewPos - FragPos);
	vec3 reflectDir = reflect(-lightDir, norm);
	float spec = pow(max(dot(viewDir, reflectDir), 0), material.shininess);
	vec3 specularLight = spec * material.specular * light.color * light.specular;

	color = vec4((ambientLight + diffuseLight + specularLight), 1.0f);
};
```

Code in C++

Application.cpp

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <iostream>

//#define GLM_FORCE_CTOR_INIT

#include "Renderer.h"
#include "VertexBuffer.h"
//#include "IndexBuffer.h"
#include "VertexArray.h"
#include "VertexBufferLayout.h"
#include "Shader.h"
//#include "Texture.h"
#include "Camera.h"
//#include "Material.h"
#include "Light.h"

#include "glm/glm.hpp"
#include "glm/gtc/matrix_transform.hpp"
#include <glm/gtc/type_ptr.hpp>

#include <glm/gtx/string_cast.hpp>
#include "Model.h"



#define WIDTH 1920
#define HEIGHT 1080



GLFWwindow* window;
```

```
int initialization();

void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode);
void do_movement();

void mouse_callback(GLFWwindow* window, double xpos, double ypos);

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset);

void set_camera(glm::vec3 carPos, float theta);



const glm::vec3 cameraPos = glm::vec3(-50.0f, 50.0f, 50.0f);
const glm::vec3 cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
//const glm::vec3 cameraFront = glm::vec3(1.0f, 0.0f, 0.0f);
const glm::vec3 worldUp = glm::vec3(0.0f, 1.0f, 0.0f);
const float fov = 45.0f;
const float yaw = 0.0f;
const float pitch = 0.0f;
const float keystroke_speed = 50.0f;
const float mouse_sensitivity = 0.50f;

Camera camera(cameraPos, cameraFront, worldUp, fov, yaw, pitch, keystroke_speed, mouse_sensitivity);



bool keys[1024];
float deltaTime = 0.0f;

bool firstMouse = true;
float lastX;
float lastY;
```

```
int main(void)
{
	if (initialization() == -1)
		return -1;

	Model SunModel("res/models/sun/sun.obj");

	Shader SunShader("res/shaders/LightingSource.shader");
	glm::mat4 light_model(1.0);
	light_model = glm::translate(light_model, Lights::Sun.Position);
	light_model = glm::scale(light_model, glm::vec3(10.0f));
	SunShader.Bind();
	SunShader.SetUniformMat4f("model", light_model);
	SunShader.SetUniform3f("lightColor", Lights::Sun.Color);


	Model envModel("res/models/env/env.obj");

	float env_rotate = 0.0f;//-90.0f;
	Shader EnvShader("res/shaders/Model.shader");
	EnvShader.Bind();
	glm::mat4 env_model(1.0);
	env_model = glm::rotate(env_model, glm::radians(env_rotate), glm::vec3(0.0f, 1.0f, 0.0f));
	env_model = glm::scale(env_model, glm::vec3(5.0f, 5.0f, 5.0f));
	EnvShader.SetUniformMat4f("model", env_model);
	EnvShader.SetUniformLight("light", Lights::Sun);
	glm::mat3 transpose_inverse_env_model = glm::transpose(glm::inverse(env_model));
	EnvShader.SetUniformMat3f("transpose_inverse_model", transpose_inverse_env_model);
```

```
Model Chevrolet("res/models/chevrolet_camaro_ss/Chevrolet_Camaro_SS_High.obj");
Shader ChevroletShader("res/shaders/Model.shader");
glm::vec3 ChevroletPos(6.0f, 2.80f, -167.0f);
float ChevroletAngle = 0.0f;
glm::mat4 chevrolet_model(1.0f);
chevrolet_model = glm::translate(chevrolet_model, ChevroletPos);
ChevroletShader.Bind();
ChevroletShader.SetUniformMat4f("model", chevrolet_model);
ChevroletShader.SetUniformLight("light", Lights::Sun);
```

```
Model Camioneta("res/models/camioneta/camioneta_High.obj");
Shader CamionetaShader("res/shaders/Model.shader");
glm::vec3 CamionetaPos(6.0f, 3.35f, -88.0f);
float CamionetaAngle = 0.0f;
glm::mat4 camioneta_model(1.0f);
camioneta_model = glm::translate(camioneta_model, CamionetaPos);
CamionetaShader.Bind();
CamionetaShader.SetUniformMat4f("model", camioneta_model);
CamionetaShader.SetUniformLight("light", Lights::Sun);
```

```
Model Toyota("res/models/25-vaz-2108/Ba3 2108.obj");
Shader ToyotaShader("res/shaders/Model.shader");
glm::vec3 ToyotaPos(-8.5f, 0.35f, 167.0f);
float ToyotaAngle = -90.0f;
glm::mat4 toyota_model(1.0f);
toyota_model = glm::translate(toyota_model, ToyotaPos);
```

```
toyota_model = glm::scale(toyota_model, glm::vec3(1.40f, 1.40f, 1.40f));
toyota_model = glm::rotate(toyota_model, glm::radians(ToyotaAngle), glm::vec3(0.0f, 1.0f, 0.0f));
ToyotaShader.Bind();
ToyotaShader.SetUniformMat4f("model", toyota_model);
ToyotaShader.SetUniformLight("light", Lights::Sun);




Model Bus("res/models/bus/BUS.obj");
Shader BusShader("res/shaders/Model.shader");
glm::vec3 BusPos(-11.0f, 8.35f, 127.0f);
float BusAngle = 90.0f;
glm::mat4 bus_model(1.0f);
bus_model = glm::translate(bus_model, BusPos);
bus_model = glm::scale(bus_model, glm::vec3(2.50f, 2.50f, 2.50f));
bus_model = glm::rotate(bus_model, glm::radians(BusAngle), glm::vec3(0.0f, 1.0f, 0.0f));
BusShader.Bind();
BusShader.SetUniformMat4f("model", bus_model);
BusShader.SetUniformLight("light", Lights::Sun);



Renderer renderer;


glEnable(GL_DEPTH_TEST);


float lastFrame = 0.0f;
float currentFrame = 0.0f;


float theta = 180.0f;


while (glfwWindowShouldClose(window) == 0)
{
```

```
glm::vec3 sky_color(72.0f, 119.0f, 167.0f);
renderer.Clear(sky_color);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

currentFrame = glfwGetTime();
deltaTime = currentFrame - lastFrame;
lastFrame = currentFrame;

CamionetaPos.z += deltaTime * 20.0f;
ToyotaPos.x -= deltaTime * 15.0f;
BusPos.x += deltaTime * 6.0f;

if (CamionetaPos.z >= 184.0f)
{
	camioneta_model = glm::translate(camioneta_model, glm::vec3(0.0f, 0.0f, -CamionetaPos.z -
150.0f));
	CamionetaPos.z = -150.0f;
}


if (ToyotaPos.x <= -245.0f)
{
	toyota_model = glm::translate(toyota_model, glm::vec3(-ToyotaPos.x - 0.0f, 0.0f, 0.0f));
	ToyotaPos.x = 0.0f;
}


if (BusPos.x >= 104.0f)
{
	bus_model = glm::translate(bus_model, glm::vec3(-BusPos.x -30.0f, 0.0f, 0.0f));
	BusPos.x = -30.0f;
}

std::cout << ChevroletPos.z << std::endl;
```

```
camioneta_model = glm::translate(camioneta_model, glm::vec3(0.0f, 0.0f, deltaTime * 20.0f));
toyota_model = glm::translate(toyota_model, glm::vec3(-deltaTime * 15.0f, 0.0f, 0.0f));
bus_model = glm::translate(bus_model, glm::vec3(deltaTime * 6.0f, 0.0f, 0.0f));



float distance = deltaTime * 50.0f;
float turnAngle = 1.0f;

if (keys[GLFW_KEY_UP])
{
	ChevroletPos.z -= distance * cos(glm::radians(theta));
	ChevroletPos.x -= distance * sin(glm::radians(theta));
	if (abs(ChevroletPos.x) < 8.50f && abs(ChevroletPos.z) < 170.0f)
		chevrolet_model = glm::translate(chevrolet_model,
glm::vec3(sin(glm::radians(ChevroletAngle)) * distance, 0.0f, /*ChevroletAngle / abs(ChevroletAngle) **/
cos(glm::radians(ChevroletAngle)) * distance));
	else
	{
		ChevroletPos.z += distance * cos(glm::radians(theta));
		ChevroletPos.x += distance * sin(glm::radians(theta));
	}
}
if (keys[GLFW_KEY_LEFT])
{
	theta += turnAngle;
	chevrolet_model = glm::rotate(chevrolet_model, glm::radians(turnAngle), glm::vec3(0.0f,
1.0f, 0.0f));
}
if (keys[GLFW_KEY_RIGHT])
{
	theta -= turnAngle;
	chevrolet_model = glm::rotate(chevrolet_model, glm::radians(-turnAngle), glm::vec3(0.0f,
1.0f, 0.0f));
```

```
		}
		if (keys[GLFW_KEY_DOWN])
		{
			ChevroletPos.z += distance * cos(glm::radians(theta));
			ChevroletPos.x += distance * sin(glm::radians(theta));
			if (abs(ChevroletPos.x) < 8.50f && abs(ChevroletPos.z) < 170.0f)
				chevrolet_model = glm::translate(chevrolet_model, glm::vec3(-
sin(glm::radians(ChevroletAngle)) * distance, 0.0f,/* ChevroletAngle /abs(ChevroletAngle)* */ -
cos(glm::radians(ChevroletAngle)) * distance));
			else
			{
				ChevroletPos.z -= distance * cos(glm::radians(theta));
				ChevroletPos.x -= distance * sin(glm::radians(theta));
			}
		}

		if (keys[GLFW_KEY_PAGE_UP])
		{
			ChevroletPos.y += 0.1f;
			chevrolet_model = glm::translate(chevrolet_model, glm::vec3(0.0f, 0.1f, 0.0f));
		}

		if (keys[GLFW_KEY_PAGE_DOWN])
		{
			ChevroletPos.y -= 0.1f;
			chevrolet_model = glm::translate(chevrolet_model, glm::vec3(0.0f, -0.1f, 0.0f));
		}

		std::cout <<glm::to_string(ChevroletPos) << std::endl;

		ChevroletShader.Bind();
		ChevroletShader.SetUniformMat4f("model", chevrolet_model);
		ChevroletShader.SetUniformMat3f("transpose_inverse_model",
glm::transpose(glm::inverse(chevrolet_model)));
```

```
		CamionetaShader.Bind();
		CamionetaShader.SetUniformMat4f("model", camioneta_model);
		CamionetaShader.SetUniformMat3f("transpose_inverse_model",
glm::transpose(glm::inverse(camioneta_model)));

		ToyotaShader.Bind();
		ToyotaShader.SetUniformMat4f("model", toyota_model);
		ToyotaShader.SetUniformMat3f("transpose_inverse_model",
glm::transpose(glm::inverse(toyota_model)));

		BusShader.Bind();
		BusShader.SetUniformMat4f("model", bus_model);
		BusShader.SetUniformMat3f("transpose_inverse_model", glm::transpose(glm::inverse(bus_model)));

		glm::mat4 view = camera.GetViewMatrix();
		glm::mat4 projection = glm::perspective(glm::radians(camera.Fov), (float)WIDTH / (float)HEIGHT,
0.1f, 500.0f);

		SunShader.Bind();
		SunShader.SetUniformMat4f("view", view);
		SunShader.SetUniformMat4f("projection", projection);

		glm::vec3 viewPos = camera.Position + camera.Front;

		EnvShader.Bind();
		EnvShader.SetUniform3f("viewPos", viewPos);
		EnvShader.SetUniformMat4f("view", view);
		EnvShader.SetUniformMat4f("projection", projection);


		ChevroletShader.Bind();
		ChevroletShader.SetUniform3f("viewPos", viewPos);
		ChevroletShader.SetUniformMat4f("view", view);
		ChevroletShader.SetUniformMat4f("projection", projection);
```

```
CamionetaShader.Bind();
CamionetaShader.SetUniform3f("viewPos", viewPos);
CamionetaShader.SetUniformMat4f("view", view);
CamionetaShader.SetUniformMat4f("projection", projection);

ToyotaShader.Bind();
ToyotaShader.SetUniform3f("viewPos", viewPos);
ToyotaShader.SetUniformMat4f("view", view);
ToyotaShader.SetUniformMat4f("projection", projection);

BusShader.Bind();
BusShader.SetUniform3f("viewPos", viewPos);
BusShader.SetUniformMat4f("view", view);
BusShader.SetUniformMat4f("projection", projection);

SunModel.Draw(SunShader);
envModel.Draw(EnvShader);
Chevrolet.Draw(ChevroletShader);
Camioneta.Draw(CamionetaShader);
Toyota.Draw(ToyotaShader);
Bus.Draw(BusShader);



if (keys[GLFW_KEY_ENTER])
        camera.mouse_enable = false;
else if (keys[GLFW_KEY_M])
        camera.mouse_enable = true;
if (camera.mouse_enable)
        do_movement();
else
        set_camera(ChevroletPos, theta);

if (keys[GLFW_KEY_ESCAPE])
        glfwSetWindowShouldClose(window, GL_TRUE);
```

```
		glfwSwapBuffers(window);
		glfwPollEvents();
	}

	glfwTerminate();
	return 0;
}


int initialization()
{
	if (!glfwInit())
	{
		std::cout << "Error initializing glfw" << std::endl;
		return -1;
	}

	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	glfwWindowHint(GLFW_OPENGL_DEBUG_CONTEXT, GL_TRUE);

	window = glfwCreateWindow(WIDTH, HEIGHT, "CAR COLLISION SIMULATION",
glfwGetPrimaryMonitor(), NULL);
	if (!window)
	{
		std::cout << "Self Driving Car Simulator" << std::endl;
		glfwTerminate();
		return -1;
	}
	glfwMakeContextCurrent(window);   // make current window the Opengl rendering context

	glfwSwapInterval(1);          //synchronizing with monitor frequency i.e 60Hz
```

```
	if (glewInit() != GLEW_OK)		// glewInit() should be called after a valid OpenGL rendering context is created
		std::cout << "Error while initializing glew" << std::endl;

	glEnable(GL_DEBUG_OUTPUT);
	glDebugMessageCallback(MessageCallback, 0);

	std::cout << glGetString(GL_VERSION) << std::endl;	// print opengl version and graphics card version

	glfwSetKeyCallback(window, key_callback);

	glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_DISABLED);

	glfwSetCursorPosCallback(window, mouse_callback);

	glfwSetScrollCallback(window, scroll_callback);

	return 0;
}



void key_callback(GLFWwindow* window, int key, int scancode, int action, int mode)
{
	if (action == GLFW_PRESS)
		keys[key] = true;
	else if (action == GLFW_RELEASE)
		keys[key] = false;
}



void do_movement()
{
	if (keys[GLFW_KEY_W])
		camera.ProcessKeyboard(Camera_Movement::forward, deltaTime);
```

```
    if (keys[GLFW_KEY_S])
            camera.ProcessKeyboard(Camera_Movement::backward, deltaTime);
    if (keys[GLFW_KEY_A])
            camera.ProcessKeyboard(Camera_Movement::left, deltaTime);
    if (keys[GLFW_KEY_D])
            camera.ProcessKeyboard(Camera_Movement::right, deltaTime);
}

void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    if (firstMouse)
    {
            lastX = xpos;
            lastY = ypos;
            firstMouse = false;
    }

    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;

    lastX = xpos;
    lastY = ypos;

    if(camera.mouse_enable)
            camera.ProcessMouseMovement(xoffset, yoffset);

}

void scroll_callback(GLFWwindow* window, double xoffset, double yoffset)
{
    camera.ProcessMouseScroll(yoffset);
}

void set_camera(glm::vec3 carPos, float theta)
```

```
    {
        float camera_distance = 15.0f;
        float camera_height = 5.0f;
        camera.Position.x = carPos.x + camera_distance * sin(glm::radians(theta));
        camera.Position.y = carPos.y + camera_height;
        camera.Position.z = carPos.z + camera_distance * cos(glm::radians(theta));

        camera.Front = carPos;
    }
```

Camera.h

```
#pragma once

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

enum class Camera_Movement
{
	forward,
	backward,
	left,
	right
};

class Camera
{
private:
	const float KEYSTROKE_SPEED;
	const float MOUSE_SENSITIVITY;

	float Yaw;


	glm::vec3 WorldUp;
	glm::vec3 Up;
	glm::vec3 Right;

public:
	float Pitch;

	float Fov;
	glm::vec3 Position;
	glm::vec3 Front;

	bool mouse_enable;
```

```
public:
	Camera(glm::vec3 position, glm::vec3 front, glm::vec3 world_up, float fov, float
yaw, float pitch, float keystroke_speed, float mouse_sensitivity);

	glm::mat4 GetViewMatrix();

	void ProcessKeyboard(Camera_Movement direction, float deltaTime);

	void ProcessMouseMovement(float xoffset, float yoffset, bool constrainPitch =
true);

	void ProcessMouseScroll(float yoffset);

private:
	void UpdateCameraVectors();

    };
```

Camera.cpp

```
#include "Camera.h"

Camera::Camera(glm::vec3 position, glm::vec3 front, glm::vec3 world_up, float fov, float
yaw, float pitch, float keystroke_speed, float mouse_sensitivity)

	:KEYSTROKE_SPEED(keystroke_speed), MOUSE_SENSITIVITY(mouse_sensitivity),

	Yaw(yaw), Pitch(pitch),

	Position(position), Front(front), WorldUp(world_up), Up(1.0), Right(1.0),

	Fov(fov),

	mouse_enable(true)
{
	UpdateCameraVectors();
}

glm::mat4 Camera::GetViewMatrix()
{
	UpdateCameraVectors();
	return glm::lookAt(Position, Position + Front, Up);
}

void Camera::ProcessKeyboard(Camera_Movement direction, float deltaTime)
{
	float velocity = KEYSTROKE_SPEED * deltaTime;
	if (direction == Camera_Movement::forward)
		Position += Front * velocity;
	if (direction == Camera_Movement::backward)
		Position -= Front * velocity;
	if (direction == Camera_Movement::left)
		Position -= Right * velocity;
```

```
	if (direction == Camera_Movement::right)
		Position += Right * velocity;
}

void Camera::ProcessMouseMovement(float xoffset, float yoffset, bool constrainPitch)
{
	xoffset *= MOUSE_SENSITIVITY;
	yoffset *= MOUSE_SENSITIVITY;

	Yaw += xoffset;
	Pitch += yoffset;

	if (constrainPitch)
	{
		if (Pitch > 89.0f)
			Pitch = 89.0f;
		else if (Pitch < -89.0f)
			Pitch = -89.0f;
	}
	UpdateCameraVectors();
}

void Camera::ProcessMouseScroll(float yoffset)
{
	Fov -= yoffset;
	if (Fov <= 1.0f)
		Fov = 1.0f;
	else if (Fov >= 45.0f)
		Fov = 45.0f;
}


void Camera::UpdateCameraVectors()
{
	if (mouse_enable)
	{
		Front.x = cos(glm::radians(Yaw)) * cos(glm::radians(Pitch));
		Front.y = sin(glm::radians(Pitch));
		Front.z = sin(glm::radians(Yaw)) * cos(glm::radians(Pitch));
		Front = glm::normalize(Front);
	}

	else
	{
		Front = Front - Position;
	}
	Right = glm::normalize(glm::cross(Front, WorldUp));
	Up = glm::normalize(glm::cross(Right, Front));
   }
```

Mesh.h

```
#pragma once

#include <string>
```

```
#include <fstream>
#include <sstream>
#include <iostream>
#include <vector>
using namespace std;
// GL Includes
#include <GL/glew.h> // Contains all the necessery OpenGL includes
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>

#include "Shader.h"

#include "Material.h"


struct Vertex {
    // Position
    glm::vec3 Position;
    // Normal
    glm::vec3 Normal;
    // TexCoords
    glm::vec2 TexCoords;
};

struct Texture {
    GLuint id;
    string type;
    aiString path;
};

class Mesh {
public:
    /*  Mesh Data  */
    vector<Vertex> vertices;
    vector<GLuint> indices;
    vector<Texture> textures;
    Material material;

    /*  Functions  */
    // Constructor
    Mesh(vector<Vertex> vertices, vector<GLuint> indices, vector<Texture> textures, Mate-
rial material)
    {
        this->vertices = vertices;
        this->indices = indices;
        this->textures = textures;
        this->material = material;

        // Now that we have all the required data, set the vertex buffers and its attrib-
ute pointers.
        this->setupMesh();
    }

    // Render the mesh
    void Draw(Shader& shader)
    {
```

```
        //std::cout << glm::to_string(this->material.Ambient) << std::endl <<
glm::to_string(this->material.Diffuse) << std::endl << glm::to_string(this->mate-
rial.Specular) << std::endl << this->material.Shininess << std::endl << std::endl <<
std::endl;

        shader.SetUniformMaterial("material", this->material);
        // Bind appropriate textures
        GLuint diffuseNr = 1;
        GLuint specularNr = 1;
        for (GLuint i = 0; i < this->textures.size(); i++)
        {
            glActiveTexture(GL_TEXTURE0 + i); // Active proper texture unit before bind-
ing
            // Retrieve texture number (the N in diffuse_textureN)
            stringstream ss;
            string number;
            string name = this->textures[i].type;
            if (name == "texture_diffuse")
                ss << diffuseNr++; // Transfer GLuint to stream
            else if (name == "texture_specular")
                ss << specularNr++; // Transfer GLuint to stream
            number = ss.str();
            // Now set the sampler to the correct texture unit
            shader.SetUniform1i((name + number).c_str(), i);
            ///glUniform1i(glGetUniformLocation(shader.Program, (name + number).c_str()),
i);
            // And finally bind the texture
            glBindTexture(GL_TEXTURE_2D, this->textures[i].id);
        }

        // Also set each mesh's shininess property to a default value (if you want you
could extend this to another mesh property and possibly change this value)
        //glUniform1f(glGetUniformLocation(shader.Program, "material.shininess"), 16.0f);
        //shader.SetUniform1f("material.shininess", 16.0f);

        // Draw mesh
        glBindVertexArray(this->VAO);
        glDrawElements(GL_TRIANGLES, this->indices.size(), GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);

        // Always good practice to set everything back to defaults once configured.
        for (GLuint i = 0; i < this->textures.size(); i++)
        {
            glActiveTexture(GL_TEXTURE0 + i);
            glBindTexture(GL_TEXTURE_2D, 0);
        }
    }

private:
    /*  Render data  */
    GLuint VAO, VBO, EBO;

    /*  Functions    */
    // Initializes all the buffer objects/arrays
    void setupMesh()
    {
        // Create buffers/arrays
        glGenVertexArrays(1, &this->VAO);
```

```
        glGenBuffers(1, &this->VBO);
        glGenBuffers(1, &this->EBO);

        glBindVertexArray(this->VAO);
        // Load data into vertex buffers
        glBindBuffer(GL_ARRAY_BUFFER, this->VBO);
        // A great thing about structs is that their memory layout is sequential for all
its items.
        // The effect is that we can simply pass a pointer to the struct and it trans-
lates perfectly to a glm::vec3/2 array which
        // again translates to 3/2 floats which translates to a byte array.
        glBufferData(GL_ARRAY_BUFFER, this->vertices.size() * sizeof(Vertex), &this->ver-
tices[0], GL_STATIC_DRAW);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, this->EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER, this->indices.size() * sizeof(GLuint),
&this->indices[0], GL_STATIC_DRAW);

        // Set the vertex attribute pointers
        // Vertex Positions
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)0);
        // Vertex Normals
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)off-
setof(Vertex, Normal));
        // Vertex Texture Coords
        glEnableVertexAttribArray(2);
        glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex), (GLvoid*)off-
setof(Vertex, TexCoords));

        glBindVertexArray(0);
    }
};
```

Model.h

```
#pragma once

// Std. Includes

#include <string>

#include <fstream>

#include <sstream>

#include <iostream>

#include <map>

#include <vector>
```

```
using namespace std;
// GL Includes
#include <GL/glew.h> // Contains all the necessery OpenGL includes
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include "stb_image/stb_image.h"
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>

#include "Mesh.h"

#include "Material.h"

#include <glm/gtx/string_cast.hpp>



Material loadMaterial(aiMaterial* mat);//////////////////////////////////////////////////////
GLint TextureFromFile(const char* path, string directory);

class Model
{
public:
    /*  Functions   */
    // Constructor, expects a filepath to a 3D model.
    Model(string path)
    {
        this->loadModel(path);
    }

    // Draws the model, and thus all its meshes
    void Draw(Shader& shader)
    {
        shader.Bind();
```

```
        for (GLuint i = 0; i < this->meshes.size(); i++)
            this->meshes[i].Draw(shader);
    }

private:
    /*  Model Data  */
    vector<Mesh> meshes;
    string directory;
    vector<Texture> textures_loaded;          // Stores all the textures loaded so far, optimization to make sure textures aren't loaded more than once.

    /*  Functions   */
    // Loads a model with supported ASSIMP extensions from file and stores the resulting meshes in the meshes vector.
    void loadModel(string path)
    {
        // Read file via ASSIMP
        Assimp::Importer importer;
        const aiScene* scene = importer.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);
        // Check for errors
        if (!scene || scene->mFlags == AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) // if is Not Zero
        {
            cout << "ERROR::ASSIMP:: " << importer.GetErrorString() << endl;
            return;
        }
        // Retrieve the directory path of the filepath
        this->directory = path.substr(0, path.find_last_of('/'));

        // Process ASSIMP's root node recursively
        this->processNode(scene->mRootNode, scene);
    }

    // Processes a node in a recursive fashion. Processes each individual mesh located at the node and repeats this process on its children nodes (if any).
    void processNode(aiNode* node, const aiScene* scene)
```

```
{
    // Process each mesh located at the current node
    for (GLuint i = 0; i < node->mNumMeshes; i++)
    {
        // The node object only contains indices to index the actual objects in the scene.
        // The scene contains all the data, node is just to keep stuff organized (like relations between nodes).
        aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        this->meshes.push_back(this->processMesh(mesh, scene));
    }
    // After we've processed all of the meshes (if any) we then recursively process each of the children nodes
    for (GLuint i = 0; i < node->mNumChildren; i++)
    {
        this->processNode(node->mChildren[i], scene);
    }

}

Mesh processMesh(aiMesh* mesh, const aiScene* scene)
{
    // Data to fill
    vector<Vertex> vertices;
    vector<GLuint> indices;
    vector<Texture> textures;

    // Walk through each of the mesh's vertices
    for (GLuint i = 0; i < mesh->mNumVertices; i++)
    {
        Vertex vertex;
        glm::vec3 vector; // We declare a placeholder vector since assimp uses its own vector class that doesn't directly convert to glm's vec3 class so we transfer the data to this placeholder glm::vec3 first.
        // Positions
        vector.x = mesh->mVertices[i].x;
        vector.y = mesh->mVertices[i].y;
        vector.z = mesh->mVertices[i].z;
        vertex.Position = vector;
```

```
        // Normals
        vector.x = mesh->mNormals[i].x;
        vector.y = mesh->mNormals[i].y;
        vector.z = mesh->mNormals[i].z;
        vertex.Normal = vector;
        // Texture Coordinates
        if (mesh->mTextureCoords[0]) // Does the mesh contain texture coordinates?
        {
            glm::vec2 vec;
            // A vertex can contain up to 8 different texture coordinates. We thus make the assumption that we won't
            // use models where a vertex can have multiple texture coordinates so we always take the first set (0).
            vec.x = mesh->mTextureCoords[0][i].x;
            vec.y = mesh->mTextureCoords[0][i].y;
            vertex.TexCoords = vec;
        }
        else
            vertex.TexCoords = glm::vec2(0.0f, 0.0f);
        vertices.push_back(vertex);
    }
    // Now wak through each of the mesh's faces (a face is a mesh its triangle) and retrieve the corresponding vertex
indices.
    for (GLuint i = 0; i < mesh->mNumFaces; i++)
    {
        aiFace face = mesh->mFaces[i];
        // Retrieve all indices of the face and store them in the indices vector
        for (GLuint j = 0; j < face.mNumIndices; j++)
            indices.push_back(face.mIndices[j]);
    }
    // Process materials
    Material _material;//////////////////////////////////////////////////////////////////////////////////
    if (mesh->mMaterialIndex >= 0)
    {
        aiMaterial* material = scene->mMaterials[mesh->mMaterialIndex];

        _material = loadMaterial(material);//////////////////////////////////////////////////////////////////////////////
```

```
        // We assume a convention for sampler names in the shaders. Each diffuse texture should be named
        // as 'texture_diffuseN' where N is a sequential number ranging from 1 to MAX_SAMPLER_NUMBER.
        // Same applies to other texture as the following list summarizes:
        // Diffuse: texture_diffuseN
        // Specular: texture_specularN
        // Normal: texture_normalN

        // 1. Diffuse maps
        vector<Texture> diffuseMaps = this->loadMaterialTextures(material, aiTextureType_DIFFUSE, "texture_diffuse");
        textures.insert(textures.end(), diffuseMaps.begin(), diffuseMaps.end());
        // 2. Specular maps
        vector<Texture> specularMaps = this->loadMaterialTextures(material, aiTextureType_SPECULAR, "texture_specular");
        textures.insert(textures.end(), specularMaps.begin(), specularMaps.end());
    }

    // Return a mesh object created from the extracted mesh data
    return Mesh(vertices, indices, textures, _material);////////////////////////////////////////////////////////////////
}

// Checks all material textures of a given type and loads the textures if they're not loaded yet.
// The required info is returned as a Texture struct.
vector<Texture> loadMaterialTextures(aiMaterial* mat, aiTextureType type, string typeName)
{
    vector<Texture> textures;
    for (GLuint i = 0; i < mat->GetTextureCount(type); i++)
    {
        aiString str;
        mat->GetTexture(type, i, &str);
        // Check if texture was loaded before and if so, continue to next iteration: skip loading a new texture
        GLboolean skip = false;
        for (GLuint j = 0; j < textures_loaded.size(); j++)
        {
```

```
            if (std::strcmp(textures_loaded[j].path.C_Str(), str.C_Str()) == 0)
            {
                textures.push_back(textures_loaded[j]);
                skip = true; // A texture with the same filepath has already been loaded, continue to next one. (optimization)
                break;
            }
        }
        if (!skip)
        {   // If texture hasn't been loaded already, load it
            Texture texture;
            texture.id = TextureFromFile(str.C_Str(), this->directory);
            texture.type = typeName;
            texture.path = str;
            textures.push_back(texture);
            this->textures_loaded.push_back(texture);  // Store it as texture loaded for entire model, to ensure we won't unnecesery load duplicate textures.
        }
    }
    return textures;
  }
};
```

```
GLint TextureFromFile(const char* path, string directory)
{
  //Generate texture ID and load texture data
  string filename = string(path);
  filename = directory + '/' + filename;
  GLuint textureID;
  glGenTextures(1, &textureID);
  int width, height;
  stbi_set_flip_vertically_on_load(1);       //flip the image vertically on loading////////////////////
```

```
    unsigned char* image = stbi_load(filename.c_str(), &width, &height, 0, 3);//////////////////
    // Assign texture to ID
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
    glGenerateMipmap(GL_TEXTURE_2D);

    // Parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glBindTexture(GL_TEXTURE_2D, 0);
    stbi_image_free(image);////////////////////////////
    return textureID;
}
//////////////////////////////////////////////////////////////////////////////////
Material loadMaterial(aiMaterial* mat) {
    Material material;
    aiColor3D color(0.f, 0.f, 0.f);
    float shininess;

    mat->Get(AI_MATKEY_COLOR_DIFFUSE, color);
    material.Diffuse = glm::vec3(color.r, color.g, color.b);

    mat->Get(AI_MATKEY_COLOR_AMBIENT, color);
    material.Ambient = glm::vec3(color.r, color.g, color.b);

    mat->Get(AI_MATKEY_COLOR_SPECULAR, color);
    material.Specular = glm::vec3(color.r, color.g, color.b);

    mat->Get(AI_MATKEY_SHININESS, shininess);
    material.Shininess = shininess;

    //std::cout << glm::to_string(material.Ambient) << std::endl << glm::to_string(material.Diffuse) << std::endl <<
glm::to_string(material.Specular) << std::endl << material.Shininess << std::endl << std::endl << std::endl;
```

```
        return material;
    }
```

Renderer.h

```
#pragma once

#include <GL/glew.h>

//#include "VertexArray.h"
//#include "IndexBuffer.h"
#include "Shader.h"

#include "glm/glm.hpp"

void GLAPIENTRY MessageCallback(GLenum source, GLenum type, GLuint id, GLenum severity,
GLsizei length, const GLchar * message, const void* userParam);

class Renderer
{
public:
       void Clear(glm::vec3& color_RGB) const;
       //void Draw(const VertexArray& va, const IndexBuffer& ib, const Shader& shader)
const;
    };
```

Renderer.cpp

```
#include "Renderer.h"

#include <iostream>

void GLAPIENTRY MessageCallback(GLenum source, GLenum type, GLuint id, GLenum severity,
GLsizei length, const GLchar* message, const void* userParam)
{
       std::cout << "OPENGL ERROR :" << std::endl << message << std::endl;
       __debugbreak();

}

void Renderer::Clear(glm::vec3& color_RGB) const
{
       glClearColor(color_RGB.x / 255.0f, color_RGB.y / 255.0f, color_RGB.z / 255.0f,
1.0f);

       //glClearColor(0.90f, 1.0f, 1.0f, 1.0f);  // celeste polvere (powdery)
```

```
	//glClearColor(0.8f, 1.0f, 1.0f, 1.0f);		// celeste pallido (pale)
	//glClearColor(0.8f, 0.9f, 0.9f, 1.0f);		// celeste velato (veiled / over-
cast)
	//glClearColor(0.7f, 1.0f, 1.0f, 1.0f);		// celeste (sky blue, heavenly
blue)
	//glClearColor(0.5f, 0.8f, 0.8f, 1.0f);		// celeste opaco (opaque)
	//glClearColor(0.0f, 0.75f, 1.0f, 1.0f);  //deep sky blue

	//glClearColor(0.45f, 0.82f, 0.98f, 1.0f);	//custom picked value


	//glClear(GL_COLOR_BUFFER_BIT);

}
/*
void Renderer::Draw(const VertexArray& va, const IndexBuffer& ib, const Shader& shader)
const
{
	va.Bind();
	ib.Bind();
	shader.Bind();

	glDrawElements(GL_TRIANGLES, ib.GetCount(), GL_UNSIGNED_INT, nullptr);
}
*/
```

Light.h

```
#pragma once

#include "glm/glm.hpp"

class Light
{
public:
	glm::vec3 Color;
	glm::vec3 Position;
	glm::vec3 Ambient;
	glm::vec3 Diffuse;
	glm::vec3 Specular;

	Light(glm::vec3 color, glm::vec3 position, glm::vec3 ambient, glm::vec3 diffuse,
glm::vec3 specular)
		:Color(color), Position(position), Ambient(ambient), Diffuse(diffuse),
Specular(specular)
	{}
};



namespace Lights
{
	static glm::vec3 Coral(0.96f, 0.53f, 0.34f);
	static glm::vec3 MellowApricot(0.94f, 0.73f, 0.44f);
	static Light Sun(Coral, glm::vec3(-100.0f, 100.0f, 200.0f), glm::vec3(0.1f, 0.1f,
0.1f), glm::vec3(1.0f, 1.0f, 1.0f), glm::vec3(1.0f, 1.0f, 1.0f));
```

```
	static Light light1(glm::vec3(0.94f, 0.73f, 0.44f), glm::vec3(-300.0f, 300.0f,
300.0f), glm::vec3(0.1f, 0.1f, 0.1f), glm::vec3(1.0f, 1.0f, 1.0f), glm::vec3(1.0f, 1.0f,
1.0f));
    }
```

Shader.h

```
#pragma once
#include <string>
#include <unordered_map>

#include "glm/glm.hpp"
#include"Light.h"
#include "Material.h"


struct ShaderProgramSource
{
	std::string VertexSource;
	std::string FragmentSource;
};


class Shader
{
private:
	std::string m_FilePath;
	unsigned int m_RendererID;
	std::unordered_map<std::string, int> m_UniformLocationCache;

public:
	Shader(const std::string& filepath);
	~Shader();

	void Bind() const;
	void Unbind() const;

	//Set Uniforms
	void SetUniform1i(const std::string& name, int value);
	void SetUniform1f(const std::string& name, float value);
	void SetUniform3f(const std::string& name, const glm::vec3& vector);
	void SetUniform4f(const std::string& name, float v0, float v1, float v2, float
v3);
	void SetUniformMat3f(const std::string& name, const glm::mat3& matrix);
	void SetUniformMat4f(const std::string& name, const glm::mat4& matrix);
	void SetUniformMaterial(const std::string& name, const Material& material);
	void SetUniformLight(const std::string& name, const Light& light);
	//void SetUniformLightingMap(const std::string& name, const LightingMap& lighting-
Map);

private:
	ShaderProgramSource ParseShader(const std::string& filepath);
	unsigned int CompileShader(unsigned int type, const std::string& source);
	unsigned int CreateShader(const std::string& vertexShader, const std::string&
fragmentShader);
	unsigned int GetUniformLocation(const std::string& name);
};
```

Shader.cpp

```
#include "Shader.h"

#include <GL\glew.h>

#include <iostream>
#include <fstream>   // file stream
#include <string>
#include <sstream>   //header providing stringstream class


Shader::Shader(const std::string& filepath)
	:m_RendererID(0), m_FilePath(filepath)
{
	ShaderProgramSource source = ParseShader(filepath);
	m_RendererID = CreateShader(source.VertexSource, source.FragmentSource);
}

Shader::~Shader()
{
	glDeleteProgram(m_RendererID);
}

ShaderProgramSource Shader::ParseShader(const std::string& filepath)
{
	std::ifstream stream(filepath);    // puts the file into the buffer with ID stream

	enum class ShaderType
	{
		NONE = -1, VERTEX = 0, FRAGMENT = 1
	};
	std::string line;
	std::stringstream ss[2];    //creating an object of stringstream class
	ShaderType type = ShaderType::NONE;
	while (getline(stream, line))
	{
		if (line.find("#shader") != std::string::npos)
		{
			if (line.find("vertex") != std::string::npos)
				type = ShaderType::VERTEX;
			else if (line.find("fragment") != std::string::npos)
				type = ShaderType::FRAGMENT;

		}
		else
		{
			ss[(int)type] << line << '\n';
		}
	}
	return { ss[0].str(), ss[1].str() };
}

unsigned int Shader::CompileShader(unsigned int type, const std::string& source)
{
	unsigned int id = glCreateShader(type);
```

```
	const char* src = source.c_str();  // same as &source[0]
	glShaderSource(id, 1, &src, nullptr);
	glCompileShader(id);

	int result;
	glGetShaderiv(id, GL_COMPILE_STATUS, &result);
	if (result == GL_FALSE)
	{
		int length;
		glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length);
		char* message = (char*)malloc(length * sizeof(char));
		glGetShaderInfoLog(id, length, &length, message);
		std::cout << "Failed to compile" << (type == GL_VERTEX_SHADER ? "vertex" :
"fragment") << "shader! of " << m_FilePath << std::endl;
		std::cout << message << std::endl;
		glDeleteShader(id);
		return 0;
	}

	return id;
}


unsigned int Shader::CreateShader(const std::string& vertexShader, const std::string&
fragmentShader)
{
	unsigned int program = glCreateProgram();
	unsigned int vs = CompileShader(GL_VERTEX_SHADER, vertexShader);
	unsigned int fs = CompileShader(GL_FRAGMENT_SHADER, fragmentShader);

	glAttachShader(program, vs);
	glAttachShader(program, fs);
	glLinkProgram(program);
	glValidateProgram(program);

	glDeleteShader(vs);
	glDeleteShader(fs);

	return program;
}


void Shader::Bind() const
{
	glUseProgram(m_RendererID);
}

void Shader::Unbind() const
{
	glUseProgram(0);
}

void Shader::SetUniform1i(const std::string& name, int value)
{
	glUniform1i(GetUniformLocation(name), value);
}

void Shader::SetUniform1f(const std::string& name, float value)
```

```
{
	glUniform1f(GetUniformLocation(name), value);
}

void Shader::SetUniform3f(const std::string& name, const glm::vec3& vector)
{
	glUniform3f(GetUniformLocation(name), vector.x, vector.y, vector.z);
}

void Shader::SetUniform4f(const std::string& name, float v0, float v1, float v2, float
v3)
{
	glUniform4f(GetUniformLocation(name), v0, v1, v2, v3);
}

void Shader::SetUniformMat3f(const std::string& name, const glm::mat3& matrix)
{
	glUniformMatrix3fv(GetUniformLocation(name), 1, GL_FALSE, &matrix[0][0]);
}

void Shader::SetUniformMat4f(const std::string& name, const glm::mat4& matrix)
{
	glUniformMatrix4fv(GetUniformLocation(name), 1, GL_FALSE, &matrix[0][0]);
}

void Shader::SetUniformMaterial(const std::string& name, const Material& material)
{
	SetUniform3f(name + ".ambient", material.Ambient);
	SetUniform3f(name + ".diffuse", material.Diffuse);
	SetUniform3f(name + ".specular", material.Specular);
	SetUniform1f(name + ".shininess", material.Shininess);
}

void Shader::SetUniformLight(const std::string& name, const Light& light)
{
	SetUniform3f(name + ".color", light.Color);
	SetUniform3f(name + ".position", light.Position);
	SetUniform3f(name + ".ambient", light.Ambient);
	SetUniform3f(name + ".diffuse", light.Diffuse);
	SetUniform3f(name + ".specular", light.Specular);
}
/*
void Shader::SetUniformLightingMap(const std::string& name, const LightingMap& lighting-
Map)
{
	SetUniform1i(name+".diffuse", lightingMap.DiffuseSlot);
	SetUniform1i(name+".specular", lightingMap.SpecularSlot);
	SetUniform1f(name + ".shininess", lightingMap.Shininess);
}
*/
unsigned int Shader::GetUniformLocation(const std::string& name)
{
	if (m_UniformLocationCache.find(name) != m_UniformLocationCache.end())
		return m_UniformLocationCache[name];

	int location = glGetUniformLocation(m_RendererID, name.c_str());
	if (location == -1)
		std::cout << "Warning: uniform '" << name << "' does not exist!";
```

```
    m_UniformLocationCache[name] = location;
    return location;
}
```