# TUTORIAL #1 OPENGL C++

## Goal

This tutorial is partially a lecture. I will introduce a few concepts surrounding OpenGL that you need to grasp before starting to type code. Then, I will go through the process of creating a basic OpenGL program in a tutorial style. If you have already done this, (e.g. you took Markus' undergraduate module) still follow along so you have the same starting codebase as everyone for your assignment!

The end result of this tutorial will be a working renderer that imports one obj + mtl file and rasterises them using Phong shading in one of two viewports. You will also have set up the basic codebase necessary for your raytracing assignment which will be rendered on the unused viewport.

## LIBRARIES

OpenGL Is an Application Programming Interface (API) to manipulate graphics and images. Actually, it is just a specification developed and maintained by the Khronos Group, an open, non-profit, member-driven consortium of 170 organizations. In fact, "OpenGL" is then implemented by the graphics cards (in C). That is why depending on your card/drivers you will support different versions of it.

Nowadays, it is an Implementation of the pipeline we described in the lectures. In the past (before 3.2) used to abstract a lot of the steps in the pipeline. These got deprecated over giving more flexibility to the programmers (shaders!). The main reason it sticks around is for being easier if you are not interested in that flexibility. Modern implementation of the pipeline such as WebGL and OpenGL ES do not include it as an option.

What doesn't it do? Creates windows, handles input, Buttons, widgets, rendering text, loading models, loading textures, matrices, audio, threading, VR, or give you information about your Graphics card capabilities. So actually, a fair amount of stuff.

So for that, there are several "bff" libraries that you should know of, even before we get started with any code.

### GLUT

It is the "OpenGL Utility Toolkit". The baseline "standard" for doing a lot of the necessary things. E.g. creating windows, handling input, some matrix calculation, etc. The problem is that it is no longer maintained (since 1998). It was developed under a license that did not allow further developing it. So while you can still find the original headers, it doesn't mean you should.

You can use freeglut, which has less bugs than the original, but stays faithful to the original GLUT so old applications can still be compiled and linked to it. One downside of it is that in order to continue to be faithful to the original, it does not support a few things that became more common after 1998 such as multiple windows and monitors for a single application.

### GLFW

The Graphics Library Framework. A common replacement for a few of the things GLUT was used for. It creates and manages windows and contexts (more about this in a bit). It supports OpenGL, GL ES, Vulkan, Extensions, which makes it attractive nowadays when you have more options than traditional OpenGL. It allows multiple windows and monitors, input via keyboard, mouse, gamepads, and other events and callbacks.

Long story short, you are able to create a minimal OpenGL application using it.

### GLEW

The OpenGL Extension Wrangler Library, besides being geniuses with their cowboy hat logo, they offer something that is very practical to programmers. Some graphic cards manufacturers implement specific "extensions" to core OpenGL to try and attract customers. They will say "optimize your game to our cards, because we have this shiny new feature that makes things look nicer", and people fall for it. At a certain point in time, these functions may not be part of the core OpenGL. But eventually they can be made permanent additions.

What GLEW does for you is asking which extensions do you have available, and set them up to be used for you. It also loads your OpenGL implementation for you, so don't forget about it. There are alternatives (glad,gl3w, etc.) but their logos are subpar.

### GLM

OpenGL Mathematics, with a very boring logo, offers support for some mathematical functions that are typically used in rendering. Example: matrices! GLSL, the programming language used to write shaders in OpenGL (which we will talk about in a bit), has matrices and some operations as part of the language. GLM offers exactly that functionality so you can use your matrices in GLSL seamlessly. It also does transformations, quaternions, random numbers, and some other things.

Why isn't this part of the standard? Well, people may be using other libraries for their maths and matrices (e.g. OpenCV). That is exactly our case! We will be using our beloved Homogeneous4, Cartesian3, Quaternion and Matrix4 classes.

### GLI

The less known cousin, "OpenGL Image", allows loading textures in a particular format that contains all the openGL information (DDS, KTX).

If you want to use a "normal" image format, use things like libpng, libjpg, etc. We are not importing textures in this tutorial so don't worry about this for now.

### IDES

I strongly recommend you set one up and learn to use it! Lab machines have VSCode (hence the logo). There are plenty resources on this, and I will not cover it here as "my favourite IDE" is sometimes as controversial as politics.

## 0) Getting Started

Now that you know what OpenGL is, and that we will be needing a few libraries to work with it, let's dive into the code and get the basic stuff I handed out to you compiling and running.

For this tutorial and assignment, we will be using premake to generate our project files (the instructions for our compiler on how to compile our project). We will do this because as we have just seen, there are a lot of dependencies that we will need to compile and link to have our basic opengl application. Namely, we are going to be using GLFW, GLEW, and some helper classes.

Open a command line window inside the folder where the source is located (where you can see the premake binaries), and try the following.

```
.\premake5.exe --help Windows
.\premake5 --help MacOS/Linux
```

This gives you some instructions on how to use it, depending on which OS you are using. To actually generate the project files, it will depend on the compiler you are going to use. So the following might be different to you, but here's what would could be the case for Windows, Linux, and MacOS:

```
.\premake5.exe vs2022
.\premake5 gmake
.\premake xcode4
```

Done! You must now have a .sln file, or a makefile, or a .xcodeproj that you can either double click to open an IDE, or write "make" on the command line to compile.

I recommend using an IDE for this, as it is always more helpful than typing things out on notepad. And you do get syntax completion, highlighting, and even integration with the compiler and debugger. The lab machines have VSCode installed. If you now try the following on a lab machine:

```
module load vscode
vscode .
```

It opens VSCode with the current folder there. You can open a terminal to compile and run things from VSCode, and also set up GDB if you want. I'll let you figure that one out as there will be great tutorials online to do help you get it set up. Your project is now set up, and if you run, nothing happens. As our main.cpp is absolutely empty! I want to go through ALL the steps that involve creating an OpenGL application with you. So buckle up, and get your Ctrl+C, Ctrl+V (or the apple equivalent) ready, and let's do it.

## 1) Main

As you well know (I expect) the entry point for a C++ application is the main function. So please open your main.cpp file so we can do some work with it.

In C++, our main procedure has two parameters:

```
int main(int argc, char **argv)
    {
```

argc tells you the number of parameters you have provided to the program, separated by spaces, and argv is an array with them. The name of the program is always the first argument.

Now for our basic initialization, we are expecting 3 parameters: Name of the program (always there), path to the model, and path to the material file (more on these later). If this is not the case, we return an error message and quit.

### Exercise #0

**Add the following lines** right at the beginning so we check how many parameters were provided.

```
      // check the args to make sure there's an input file

    if (argc != 3)
        { // bad arg count
        // print an error message
        std::cout << "Usage: " << argv[0] << " geometry material" << std::endl;
        // and leave
        return 0;
        } // bad arg count
}

//add this as an include in your main.cpp
#include <iostream>
```

Let's **run the application** to see it in action. That means either calling it in the terminal in your output directory, or hitting the run button on your IDE. The expected result is to see the error message you have added! Remember several IDEs provide the option of attaching a debugger to the execution of your program if you build it in debug mode.

## EXERCISE #1: CREATING WINDOW AND CONTEXT

OpenGL operates as a state machine defined by the current "context". It's not explicitly an "object oriented" paradigm where you set variables in an object, but in concept is a very similar thing. You create an OpenGL context (object) for your application, connect it to a window, then all the following OpenGL calls you make will operate on the active "context". These calls will be setting variables/states that will affect the next time rendering happens.

Let's start by writing a function that encapsulates all that initialization behaviour. To keep things absolutely simple, let's just slap everything on "main.cpp". Variables and functions are all local to the file.

```
using namespace std;

#include <GL/glew.h>
#include <GLFW/glfw3.h>
bool initializeGL()
{
    // Initialise GLFW
    if (!glfwInit())
    {
        cerr << "Failed to initialize GLFW" << endl;
        return false;
    }
```

We added a few includes (iostream for printing, GLFW and GLEW), and a using statement given that we will use a lot of std functions. We define our function with returning a Boolean to inform the main if initialization was okay. So we start by initializing GLFW. If it works, we can start setting things to create our window!

```
//Variables
GLFWwindow* window;
int window_width = 1920;
int window_height = 1080;
```

Add these to the beginning of your file. And then the following to your initializeGL function

```
glfwWindowHint(GLFW_SAMPLES, 1); //no anti-aliasing
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make MacOS happy;
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
window = glfwCreateWindow(window_width, window_height, "OpenGLRenderer", NULL, NULL);
```

These instructions create a window that is expecting an OpenGL context of version 4.5, no anti-aliasing, using the core profile (no compatibility with older versions through extensions), and that deprecated things are actually removed.

```
if (window == NULL) {
    cerr<< "Failed to open GLFW window. If you have an Intel GPU, they may not be 4.5 
compatible."<< endl;
    glfwTerminate();
    return false;
}
glfwMakeContextCurrent(window);
```

If the window failed to be created, it most likely means that our implementation of OpenGL told GLFW "no can't do sir, something that you set up there in your requirements is not okay for me". Which more often than not, the problem is the OpenGL version. If that is the case, we must terminate glfw. If things worked out, we will say "okay, all of my future calls will affect this one window!". This is how you can switch windows and contexts easily using GLFW.

Now let's load up the extensions and our OpenGL.

```
// Initialize GLEW
glewExperimental = true; // Needed for core profile
if (glewInit() != GLEW_OK) {
    cerr << "Failed to initialize GLEW" << endl;
    glfwTerminate();
    return false;
}
```

Similarly to glfw, we just call the initialization method. What it actually does, is it sets the values for several global constants that we can quickly check the values to evaluate what we are able to do. Comparison of how to check for the same thing with and without glew.

```
//without
int NumberOfExtensions;
glGetIntegerv(GL_NUM_EXTENSIONS, &NumberOfExtensions);
for (i = 0; i < NumberOfExtensions; i++) {
    const GLubyte* ccc = glGetStringi(GL_EXTENSIONS, i);
    if (strcmp(ccc, (const GLubyte*)"GL_ARB_debug_output") != 0) {
        return false;
    }
}
//with
if (!GLEW_ARB_debug_output) return false;
```

A lot simpler. Just add the second one instead of the first! Finally, let's check some things related to input:

```
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
glfwSetInputMode(window, GLFW_CURSOR, GLFW_CURSOR_NORMAL);
glfwPollEvents();

return true;
}
```

This is setting a few things related to keyboard and mouse event handling. “Sticky keys” being on has little to do with that annoying pop-up that comes on windows when you are hesitating to start your sentence and you hit shift 5 times in a row. What it does is that it allows a little simpler input processing. If you press a key (in our case, esc when we want to close the application) the status will be “pressed” until we check it with “glfwGetKey”, so we don’t risk missing that it was pressed. The second line is just allowing us to have a normal mouse that can click and control our camera and object. Done! Our window is initialized, so we can call this function on our main.

```
if (!initializeGL()) return -1;
```

We will be revisiting this function later to tweak a few more things. But let’s move to something else.

## 2) MODELLING

The goal of our application is to read some 3D model from the disk and render it on the screen using raytracing. For this, we need some internal representation of it in our application. Thankfully for you, I’ve wrote most of the code that takes care of it for you, with just a few bits left for us to complete now.

As we have seen during the input section of this tutorial, our parameters are two things: a 3D model, and a Material file. Let’s look at a simple example of each one of them to understand what is in there. If you already know what an obj and mtl files are, **do not skip.** These are no ordinary obj and mtl, these are my own version of them, given that the standard format does not support some of the things I wanted to include in this assignment.

Open the file triangle_backplane.obj in a text editor, such as notepad or VSCode. It starts with a few commented lines that sometimes are generated by the editing software, or typed by the author of the model:

```
# Vertices: 7
# Faces: 3
```

Then followed by a list of vertices (v) and vertex normals (vn) and texture coordinates (vt)

```
vn 0.333028 -0.090826 0.938533
v 0.000000 1.000000 0.000000
vt 1.0 0.0 0.0
```

Finally, let’s define objects.

```
usemtl brown
f 1/1/1 2/2/2 3/3/3
usemtl gray
f 5/6/5 4/1/4 6/5/6
f 6/5/6 4/1/4 7/2/7
```

This is saying:

We have a first object that will use the material “brown” that will be defined in the material file. It is consisted by one face (f) that connects three vertices. Using the syntax v/vt/vn use the ones that were defined in this order in the previous section.

This allows you to reuse vertices, normals and texture coordinates when defining triangles.

Now let’s look at the material file: triangle_backplane.mtl. Open it in a text editor as well. It defines the materials that we have seen in the .obj file. Let’s look into one of them.

```
newmtl brown
Ka  0.0800  0.0230  0.0230 #ambient rgb
```

```
Kd  0.2800  0.1430  0.1430 #diffuse rgb
Ks  0.2084  0.2084  0.2084 #specular rgb
Ke  0.0000  0.0000  0.0000 #emission rgb
Ns  1.0 #specular exponent
N_ior  1.0 # Index of refraction (for refraction/Fresnel effects, 1.0 is air)
N_mirr  0.0 # mirror. Float value 0 to 1
N_transp 0.0 # binary. Either transparent or opaque.
```

For each model provided, there is a matching material file. Now let's go back to looking at main.cpp and use some of the provided code to parse things.

We are going to be calling the relevant code in "ThreeDModel" to read all the models from inside our file, the materials, and link them up. Let's look at ThreeDModel.h. It is basically a container for all the things we read from the file (v,vn,vt, the corresponding faces, and the material). Look at Material.h now as well, and you will see it is also a similar case; reading the properties and place it in an accessible object. ReadObjectStreamMaterial is where all the reading happens. Read it if you want to, but the main point is now just using the provided functions:

## EXERCISE #2: USING PROVIDED CLASSES.

First, let's add to the top of the file the necessary includes.

```
#include <vector>
#include <fstream>
#include "ThreeDModel.h"
```

A container class (vector), a class to read files in an easy way, and our model class that will hold all of our information. Then the following goes after our parameter check in the main function.

```
std::vector<ThreeDModel> objects;
std::ifstream geometryFile(argv[1]);
std::ifstream materialFile(argv[2]);

// try reading it
if (!(geometryFile.good()) || !(materialFile.good())) {
    std::cout << "Read failed for object " << argv[1] << " or material " << argv[2] <<
std::endl;
    return 0;
} // object read failed
```

This now not only validates how many parameters we got, but if the files are the right ones. Now we are ready to call the appropriate function.

```
std::string s = argv[2];
//if is actually passing a material. This will trigger the modified obj read code.
if (s.find(".mtl") != std::string::npos) {
    objects = ThreeDModel::ReadObjectStreamMaterial(geometryFile, materialFile);
}

if (objects.size() == 0) {
    std::cout << "Read failed for object " << argv[1] << " or material " << argv[2] <<
std::endl;
```

```
    return 0;
} // object read failed
```

Done! We have all our objects in our vector. We will use this later for rendering. But for now if you run the application (with correct parameters, like objects/triangle_backplane.obj objects/triangle_backplane.mtl), nothing should happen.

## Exercise #3 :Finding lights

There is one extra thing that we need in our raytracer. Lights! All our scenes have some rectangles with a material called “light” with just emission. They are supposed to be the lightsources in our scenes. As we will eventually discuss, sometimes this is all you need for your raytracer. But it would be really useful for us to have the light positions accessible for our raytracer.

For that, I have provided a Light class that you can check. Our goal is to have these with our RenderParameters. It should have a vector of lights in the .h

```
#include <vector>
std::vector<Light*> lights;
```

And a function that finds them. It is defined it on the .h

```
    void findLights(std::vector<ThreeDModel> objects);
```

Read the implementation that is already there on the .cpp. This function assumes you have already read the objects and materials. So we just go through them and find the ones that have a “light” material. The function that does it literally just checks if the name contains the word light (very advanced, I know).

We have two cases to go through then: the case where we have a rectangular area light, and when our light source has a different shape. This is the non-rectangular case.

```
else
{
    Cartesian3 center = Cartesian3(0,0,0);
    for (unsigned int i = 0; i < obj.vertices.size(); i++)
    {
        center = center + obj.vertices[i];
    }
    center = center / obj.vertices.size();
    Light *l = new Light(Light::Point,obj.material->emissive,
                         center,Homogeneous4(),Homogeneous4(),Homogeneous4());
    l->enabled = true;
    lights.push_back(l);
}
```

In this case, we can only assume we have a “point light”. So we just find the center of the object and set that as the position. But in actual fact, let’s work on the previous if, where we are dealing with a rectangular area light. We will replace the print with code.

Looking at the Light class, we know we want to find a few things: Center, direction where it is pointing to, and the size of it (two tangent vectors). The normal is just any of the normal vectors for the triangles.

So let’s start with finding a vertex that is at the corner of the quad. This is a doing a double for. For each vertex in the first triangle, we check if it is in the other. If it is, it is a diagonal. We want one that has not been “found”

```
for (unsigned int i = 0; i < 3; i++)
```

```
{
    unsigned int vid = obj.faceVertices[0][i];
    bool found = false;
    for(unsigned int j = 0; j < 3; j++)
    {
        if(vid == obj.faceVertices[1][j])
        {
            found = true;
            break;
        }
    }
}
```
Right, so vid now should be our corner vertex. So we simply get the two side edges, the rest of the information, and create our light.

```
if(!found)
{
    unsigned int id1 = obj.faceVertices[0][i];
    unsigned int id2 = obj.faceVertices[0][(i+1)%3];
    unsigned int id3 = obj.faceVertices[0][(i+2)%3];
    Cartesian3 v1 = obj.vertices[id1];
    Cartesian3 v2 = obj.vertices[id2];
    Cartesian3 v3 = obj.vertices[id3];
    Cartesian3 vecA = v2 - v1;
    Cartesian3 vecB = v3 - v1;
    Homogeneous4 color = obj.material->emissive;
    Homogeneous4 pos = v1 + (vecA/2) + (vecB/2);
    Homogeneous4 normal = obj.normals[obj.faceNormals[0][0]];
    Light *l = new Light(Light::Area,color,pos,normal,vecA,vecB);
    l->enabled = true;
    lights.push_back(l);
}
```
And done! This should set all the lights in our RenderParameters. Now for the **actual exercise**, Let's call it in our main after we declare our RenderParameters, and with a little print to see if it worked.

```
RenderParameters renderParameters; //can be at file level.

renderParameters.findLights(objects);
std::cout << renderParameters.lights.size() << std::endl;
```

## 3) HELPER CLASSES

We will more than just rendering code to have this thing working, so here is a quick overview of some of the classes provided:

CARTESIAN3: a 3D vector class. Defines most of the operations you will need. Scalar operations such as multiplication and division, vector addition, dot and cross product, comparison, normalisation, etc.

HOMOGENEOUS4: a vector in Homogeneous coordinates. It has the basic operators implemented, has a "modulate" function which multiplies each element individually, and has a "vector" and "point" methods, which transforms to a Cartesian3 by either dividing by w, or dropping it.

QUATERNION: An implementation of a quaternion to be used with the arcball which is used to control rotations in this program. Take a look at the "act" method to see how it is used.

ARCBALL/ARCBALLWIDGET: A rotation controller implemented with quaternions.

MATRIX4: Has all the matrix operations you will need for this assignment. Including setting transformation matrices from vectors. Be careful as these methods overwrite the existing matrix, so they are only useful for initializing! See the implementation of "SetTranslation" as an example.

RGBAIMAGE: An image class to store the result of the calculations of our raytracer. It doesn't have a lot of functions, but hey it can output to a PPM, so could be useful for printscreens.

## EXERCISE #4: RAY CLASS PROTOTYPE

Let's create one more helper class. A representation for rays.

If you're on an IDE, right click your project and click "Add new", choose C++ class, and call it ray. You will see that files "ray.h" and "ray.cpp" are going to be created. Make them have capital R so we they are similar to all the other ones. Otherwise, just create these two new files like a normal person. But then add a few things that make a class a class.

Open the .h file and let's add a couple of public variables with the main things we want to store, and a constructor that get's those as a parameter.

```
#ifndef RAY_H
#define RAY_H
#include "Cartesian3.h"
class Ray
{
public:
    Ray(Cartesian3 og,Cartesian3 dir);
    Cartesian3 origin;
    Cartesian3 direction;
};
#endif // RAY_H
```

And let's define this constructor in the .cpp file

```
#include "Ray.h"

Ray::Ray(Cartesian3 og, Cartesian3 dir)
{
    origin = og;
    direction = dir;
}
```

This is pretty much it. We will be using this class again with the raytracing section. But first, let's look at the basic OpenGL rendering that should happen on the left window.

## 4) OpenGL Setup

### Exercise #5: Loading a Model

The data structure we loaded up there (ThreeDModel) works well for our raytracing. However, OpenGL doesn't know anything about it. Let's write a few functions to help us with that. First, a helper function that just creates multiple arrays of floats with our data: vertices, uvs, and normal.

```
//add this include
#include <array>

void ThreeDToGL(const ThreeDModel& model,
vector<array<float,3>>& out_vertices,
vector<array<float,2>>& out_uvs,
vector<array<float,3>>& out_normals)
{
for (unsigned int face = 0; face < model.faceVertices.size(); face++)
        {
            for (unsigned int triangle = 0; triangle < model.faceVertices[face].size() - 2;
             triangle++)
            {
                for (unsigned int vertex = 0; vertex < 3; vertex++)
                {
                unsigned int faceVertex = 0;
                    if (vertex != 0)
                        faceVertex = triangle + vertex;
```

Because the .obj file format accepts faces with more than 3 vertices, it's a little more complicated. FaceVertices is an array of faces. Each face is a list of vertices. We can deal with them as a triangle fan. So we will go through the faces, and get the vertices in an order that makes them into individual triangles. So for each face, we will go through its triangles, and then go through the vertices that make it, and add each one of them to our newly created "Triangle" object. The indexing is a little bit tricky, so I'll suggest you draw a little face with 5 vertices on your notebook, and follow this for loop quickly to see how it works!

If you did that, or you did not but you trust me, you will see that we are now getting each vertex for each triangle.

```
                 out_normals.push_back(array<float, 3>{
                    model.normals[model.faceNormals[face][faceVertex]].x,
                    model.normals[model.faceNormals[face][faceVertex]].y,
                    model.normals[model.faceNormals[face][faceVertex]].z
                });
                out_uvs.push_back(array<float, 2>{
                    model.textureCoords[model.faceTexCoords[face][faceVertex]].x,
                    model.textureCoords[model.faceTexCoords[face][faceVertex]].y
                });
                out_vertices.push_back(array<float, 3>{
                    model.vertices[model.faceVertices[face][faceVertex]].x,
                    model.vertices[model.faceVertices[face][faceVertex]].y,
                    model.vertices[model.faceVertices[face][faceVertex]].z
                });
```

```
            } // per vertex
        } // per triangle
    } // per face
}
```

Now if you noticed, these were just individual points, uvs, and normals. We can just specify vertices in a particular order and trust OpenGL to treat them as triangles if we want to.

Alternatively, we could use that to do what is called “indexed rendering”. We specify vertex indices from our array in the correct order for the type of primitive we want. This has lots of speed advantages, as we end up specifying less vertices, thus running our shader less times. Your next tutorial should have something like this, so we will keep it simple here.

Now we have all the information we need to render. Let’s prepare the data structures that OpenGL needs to be able to render our plane. Let’s create the variables that will be populated In our next function in our main, and call it:

```
std::vector<GLuint> vaoIDS;
std::vector<GLuint> vbIDS;
std::vector<GLuint> nbIDS;
std::vector<GLuint> tbIDS;
std::vector<unsigned int> counts;

//setting up opengl
loadModelGL(objects, vaoIDS, vbIDS, nbIDS, tbIDS, counts);
```

If you inspect what GLuint is, it’s nothing but an unsigned int. We are going to be using functions which will create resources in our OpenGL context and give us its “name”, which is a number. Let’s see what are all of these things that we are passing as parameter to be initialized while we write the function.

```
void loadModelGL(const std::vector<ThreeDModel>& objects,
    std::vector<GLuint>& vaoIDS,
    std::vector<GLuint>& vbIDS,
    std::vector<GLuint>& nbIDS,
    std::vector<GLuint>& tbIDS,
    std::vector<unsigned int>& count)
{
    for (const auto& to : objects)
    {
        GLuint VertexArrayID;
        GLuint vertexbuffer;
        GLuint uvbuffer;
        GLuint normalbuffer;
```

Each object will have four identifiers. The first one is the “Vertex Array Object”, which is the data structure we will be using to store the all the information about our model. We use glGenVertexArrays to create it.

```
         glGenVertexArrays(1, &VertexArrayID);
        glBindVertexArray(VertexArrayID);
```

The second call, glBindVertexArray will then say that all the following calls that should affect a VAO should affect the one with the given name. This is the general workflow of all things OpenGL, as we have already seen with the context. Now let’s give it the data! We start by calling the function we defined up there:

```
        std::vector<std::array<float,3>> n;
        std::vector<std::array<float, 2>> t;
        std::vector<std::array<float, 3>> v;
        ThreeDToGL(to, v, t, n);
```

Now we have this in a simple format that we can give straight to OpenGL. We are using std containers (vectors and arrays), but it is quite common to use glm for these purposes.

```
glEnableVertexAttribArray(0);
        glGenBuffers(1, &vertexbuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glBufferData(GL_ARRAY_BUFFER, v.size() * sizeof(std::array<float, 3>),
                     &v[0], GL_STATIC_DRAW);
        glVertexAttribPointer(
            0,                  // attribute
            3,                  // size (we have x y z)
            GL_FLOAT,           // type of each individual element
            GL_FALSE,           // normalized
            0,                  // stride
            (void*)0            // array buffer offset
        );
```

The first attribute we are going to be describing are our vertex positions. So we enable a first attribute at index 0, generate a new “buffer”, and then say we are going to use it in our “GL_ARRAY_BUFFER”, the source for vertex data. We then describe the data in this buffer by saying its size in bytes, provide a pointer to it, and also say that it is STATIC, meaning we won’t be changing it. This allows some optimizations. Then finally, the connection is made when we call glVertexAttribPointer, with the parameters described in the comments. If we had everything in the same memory block due to something like a direct read from a file, we could use the strides and sizes to use a single buffer to connect to multiple attributes. But it is not our case, so let’s do the same for the UVs and normals.

```
glEnableVertexAttribArray(1);
glGenBuffers(1, &uvbuffer);
glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
glBufferData(GL_ARRAY_BUFFER, t.size() * sizeof(std::array<float, 2>), &t[0],
             GL_STATIC_DRAW);
glVertexAttribPointer(1,2,GL_FLOAT,GL_FALSE,0,(void*)0);

glEnableVertexAttribArray(2);
glGenBuffers(1, &normalbuffer);
glBindBuffer(GL_ARRAY_BUFFER, normalbuffer);
glBufferData(GL_ARRAY_BUFFER, n.size() * sizeof(std::array<float, 3>), &n[0],
GL_STATIC_DRAW);
glVertexAttribPointer(2,3,GL_FLOAT,GL_FALSE,0,(void*)0);
```

The code is the same, except we are using vec2’s for the uvs since the size is different.

That is it! Now whenever we want to draw we just need to use VertexArrayID. We don’t need the original std::vectors, as we have handed over the data to OpenGL. Let’s just save the number of indices we have in our plane, as that is required for rendering later.

```
        vaoIDS.push_back(VertexArrayID);
        vbIDS.push_back(vertexbuffer);
        tbIDS.push_back(uvbuffer);
        nbIDS.push_back(normalbuffer);
        count.push_back(GLuint(v.size()));
    }
}
```

## 5) Shaders in GLSL

So we got a window, and multiple models. Next thing is writing our shaders that will do the basic rendering. Let’s talk a little big about GLSL and how it works before we go and do what we need to do in practice.

GL Shader Language, is how we are going to write these programs. It looks like c, but then also has some added functionalities to allow you to handle your usual multidimensional types a bit more easily. Let’s not worry too much about it before we start, as there will be chances to explain as we go.

Our program will be drawing our plane, and mapping a texture to it, which is fairly straightforward. Let’s assume we will also have a camera that we can move around the object. As we talked about during the lectures, we need a Vertex shader, which is always a mandatory step, and we will be writing a simple fragment shader to map the texture to the object.

### Exercise #6: Vertex Shader

The most common way to use shaders is to compile them in real time. This makes sense if you think about it: things will work depending on your graphics cards capabilities, and some of the stuff we were checking with GLEW. Same code may compile in one application and not in a different one. Some code may only be valid if you have a 4.5 context! So let’s create two files: “Basic.vert” and “Phong

.frag”. Place them on the same folder you have the premake and the textures, as that is set up to be the project’s running directory and it will make it easy for our code to find it (lazy, I know). If you are using Visual studio, drag them to your “main” project so you can double click them to edit easily. If you are using VSCode they are probably already there. There are extensions such as the “GLSL language integration” that you can install to allow syntax highlighting to happen. If you can install them, I recommend it. The extensions on the files in general don’t matter, except if you are using these packages with syntax highlighting which will only work on specific extensions (I’m using the right ones).

Let’s start by our vertex shader.

```
#version 330 core
```

We are saying our shader only requires version 330 core. This is different from our 4.5 context, but compatible. This directive exists in case you write a shader that relies in some new functionality, it will only compile if the context is correctly set. In a Tesselation shader, as an example, we would need to have a higher version here!

```
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;
```

```
// Output data ; will be interpolated for each fragment.
out vec2 UV;
out vec4 vertexPosition_cameraspace;
out vec3 EyeDirection_cameraspace;
out vec3 Normal_cameraspace;


// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 V;
uniform mat4 M;
```

So remember we set a VAO with positions and UVs? Remember that the position was on "VertexAttribArray(0)" and UV on 1? That's what we are writing here in the first two lines. We are expecting a vec3 at position 0, and a vec2 at position 1 of our input VAO. Remember that the vertex shader is a program that runs for each vertex of our model. So all of that work we had during the set up of our model saying "each element has this size, we have X number of elements, yadayada" was to allow openGL to now pack things correctly and call the vertex shader with the correct parameters fished from your buffer!

Output data is simple. It is mandatory that we always write to "gl_Position", which is what will be passed forward to the fixed part of our pipeline, so we don't need to declare that output. For our shader, we want to be also attaching UV coordinates, normal, and eye direction (for shading) to our vertex. Even if you don't plan to do anything to it, you need to declare all the outputs from your shader, and they must match the inputs of the next shader in your program, and you need to write the code that passes them forward. This is the case because you might just need a certain input parameter in the vertex shader, so after you use it, it would be a waste to "automatically" pass if forward.

Finally, in order to transform our vertex from OCS to WCS to VCS and CCS which is the coordinate system required by the next steps (e.g. clipping), we need the Model, View, and Projection matrices. We also have a single matrix parameter that is MVP, all of them combined to go straight to CCS, which saves us the hassle of doing the multiplication per vertex. Let's jump to our main function to write out what we need.

```
void main(){
    // Output position of the vertex, in clip space : MVP * position
    gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
    // UV of the vertex. No special space for this one.
    UV = vertexUV;
    // model to camera = ModelView
    Normal_cameraspace = normalize((V*M * vec4(vertexNormal_modelspace,0)).xyz);

    // Position of the vertex, in worldspace : M * position
    vertexPosition_cameraspace = ( V * M * vec4(vertexPosition_modelspace,1));
    //Eye vector. Eye is at 0 given this is camera space.
    EyeDirection_cameraspace= normalize((vec4(0,0,0,1) -
                                         vertexPosition_cameraspace).xyz);
}
```

First, we are sending in positions as 3D vectors in the object coordinate system, and we want to multiply them by the MVP matrix. Since these should be homogeneous coordinates, we append a "1" to our input position, and create a "vec4". Result gets assigned to gl_Position. We pass forward our UVs as they are, and then the normals. We are going to be calculating our shading in VCS (camera space), as it is what we will also do in the raytracing to simplify things. So we apply the view and model matrices to our normal, but make sure to have the w component as 0. Why this?

If you remember your lectures about homogeneous coordinates, the translation is described in the fourth column of the matrix. If the w coordinate is 0, when the matrix is multiplied by it, this column will have no effect. Given that the normal vectors are defined locally in relation to the surface, we just want to rotate them.

Now, for our Phong shading, lets pre-calculate the eye direction here and pass to the fragment shader to save us doing it per pixel over there. We transform the vertex to VCS, then subtract it from the camera position (zero in VCS). And that’s it! Let’s move to our fragment shader.

## EXERCISE #7: FRAGMENT SHADER.

```
#version 330 core

// Interpolated values from the vertex shaders
in vec2 UV;
in vec4 vertexPosition_cameraspace;
in vec3 EyeDirection_cameraspace;
in vec3 Normal_cameraspace;
// Ouput data
out vec3 color;
```

Notice the input matches the name of the output from the previous shader. Our output will just RGB, as we are not planning on writing anything to the alpha channel. Then, we declare all the uniforms that we need in this program. Which are quite a few.

```
struct Material{
     vec3 emissiveColor;
     vec3 diffuseColor;
     vec3 ambientColor;
     vec3 specularColor;
     float shininess;
};

struct PointLight{
    vec3 position;
    vec3 color;
};
// Values that stay constant for the whole mesh.

uniform mat4 V;
uniform mat4 M;
uniform Material meshMaterial;
#define MAX_NR_POINT_LIGHTS 3

uniform PointLight[MAX_NR_POINT_LIGHTS] lights;
uniform int nLights;
```

For our rendering we will need two structures: a material with the properties for this object, and a light with position and colour. We declare our matrices here again as we will need to transform the lights. Add the material for this specific object, and an array of lights. They need to have a fixed size, so we define it as 3 (I did not put more than 3 in the files I hand out to you), and add a parameter to let us know how many lights we actually use.

Now onto our main function:

```
void main(){
   color = meshMaterial.emissiveColor;

    for(int i = 0; i < nLights; i++)
    {
        vec3 n = normalize(Normal_cameraspace);

        vec4 lp = V*M*vec4(lights[i].position,1);
        vec3 l =normalize(lp.xyz/lp.w -
                vertexPosition_cameraspace.xyz/vertexPosition_cameraspace.w);
        vec3 e = normalize(EyeDirection_cameraspace);
```

For each one of our lights, we will be calculate the shading using the physically correct phong model. For that we need normals, an eye vector, and a light vector. Our normal and eye direction have already been transformed, so we just have to normalize them. The light vector is the light position – the vertex position. After transforming things to the proper coordinate system, we also transform them into ordinary cartesian vectors by dividing by the w coordinate.

```
//Diffuse
float cosTheta = clamp( dot( n,l ), 0,1 );
vec3 diffuse = meshMaterial.diffuseColor * lights[i].color  * cosTheta ;
vec3 ambient = meshMaterial.ambientColor * lights[i].color;
//Specular
vec3 B = normalize(l + e);
float cosB = clamp(dot(n,B),0,1);
cosB = clamp(pow(cosB,meshMaterial.shininess),0,1);
cosB = cosB * cosTheta * (meshMaterial.shininess+2)/(2*radians(180.0f));
vec3 specular = meshMaterial.specularColor* lights[i].color *cosB;
```

Nothing out of the ordinary here, just the implementation of the model as you have seen in the lectures. A few notes: clamp ensures the result is between 0 and 1, and 2*radians(180.0f) is our pi.

```
   color = color +ambient +diffuse + specular;
   }
}
```

That's it! We now accumulate the ambient, diffuse and specular per light. And our shader is done.

## EXERCISE #8: COMPILING SHADERS

Our shaders are ready to be used. As I mentioned, we need to compile them in runtime. There are other options (which we discuss in the lectures), but this is the most usual case.

Let's write first a single function that reads the shader file and compiles it, as we will have two, so this function will be called twice.

```
bool readAndCompileShader(const char* shader_path, const GLuint& id) {
```

We are assuming that a "shader object" has been created in the context, and will be passed as a parameter to this function. We will get to it in a little while.

```
#include <fstream>
#include <sstream
```

C++ way of reading files, so add these includes before the next section

```
    string shaderCode;
    ifstream shaderStream(shader_path, std::ios::in);
    if (shaderStream.is_open()) {
        stringstream sstr;
        sstr << shaderStream.rdbuf();
        shaderCode = sstr.str();
        shaderStream.close();
    }
    else {
        cout << "Impossible to open "<< shader_path << ". Are you in the right 
directory?"<< endl;
        return false;
    }
```

This part is nothing but opening a file using an “ifstream” (input file stream). We create the object (which opens the file), and if successful, we read its contents, and put it in a string. Pretty straightforward. Now, let’s compile it.

```
cout << "Compiling shader :"<< shader_path<<endl;
char const* sourcePointer = shaderCode.c_str();
glShaderSource(id, 1, &sourcePointer, NULL);
glCompileShader(id);
```

Pass the name of the shader object, number of “string” in our pointer, and then an array of lengths of each string. To keep it simple, we just have one string, and by saying “NULL” on the last parameter, we just say it is null-terminated (standard C string). Now let’s see if it went okay.

```
GLint Result = GL_FALSE;

int InfoLogLength;
glGetShaderiv(id, GL_COMPILE_STATUS, &Result);
glGetShaderiv(id, GL_INFO_LOG_LENGTH, &InfoLogLength);
if (InfoLogLength > 0) {
    vector<char> shaderErrorMessage(InfoLogLength + 1);
    glGetShaderInfoLog(id, InfoLogLength, NULL, &shaderErrorMessage[0]);
    cout << &shaderErrorMessage[0] << endl;
}
cout << "Compilation of Shader: " << shader_path << " " << (Result == GL_TRUE ? "Success" : 
"Failed!") << endl;
return Result == 1;
}
```

glGetShaderiv gives us the values for some integer properties; the results, and the length of the “info log” that was resulting of the compilation. If any information came back, we print it, then just report the status of the compilation, and exit the function. Let’s see it in use:

```
void LoadShaders(GLuint& program, const char* vertex_file_path, const char* 
fragment_file_path)
{
    // Create the shaders - tasks 1 and 2
    GLuint VertexShaderID = glCreateShader(GL_VERTEX_SHADER);
    GLuint FragmentShaderID = glCreateShader(GL_FRAGMENT_SHADER);
    bool vok = readAndCompileShader(vertex_file_path, VertexShaderID);
    bool fok = readAndCompileShader(fragment_file_path, FragmentShaderID);
```

glCreateShader creates a new shader in the context, and gives it a name. Then we just call our function and check if the compilation was okay.

```
if (vok && fok) {
        GLint Result = GL_FALSE;
        int InfoLogLength;

        cout <<"Linking program"<<endl;
        program = glCreateProgram();
        glAttachShader(program, VertexShaderID);
        glAttachShader(program, FragmentShaderID);
        glLinkProgram(program);

        glGetProgramiv(program, GL_LINK_STATUS, &Result);
        glGetProgramiv(program, GL_INFO_LOG_LENGTH, &InfoLogLength);
        if (InfoLogLength > 0) {
            std::vector<char> ProgramErrorMessage(InfoLogLength + 1);
            glGetProgramInfoLog(program, InfoLogLength, NULL, &ProgramErrorMessage[0]);
            cout << &ProgramErrorMessage[0];
        }
        std::cout << "Linking program: " << (Result == GL_TRUE ? "Success" : "Failed!") <<
std::endl;
    }else{
        std::cout << "Program will not be linked: one of the shaders has an error" <<
std::endl;
    }
```

We create a new program (assigned to our global variable), attach our two shaders to it, and then link them together. Here is where errors would show up if we don't match their expected inputs and outputs. The second sections is similar to what we do for shaders to check for errors. If no errors happened, our program is ready to be used!

```
glDeleteShader(VertexShaderID);
glDeleteShader(FragmentShaderID);
}
```

Before we exit our function, we can get rid of the shader objects, as they now have been compiled into the program.  Let's call this function inside our main:

```
GLuint programID = glCreateProgram();
LoadShaders(programID,"Basic.vert","Phong.frag");
```

## 6) Rendering

It must be complicated to just trust that all this code I'm telling you to write works, without being able to see anything. Well, let's finally move into getting things to render! The last bit that we haven't covered yet are transformation matrices and where do they come from. Our "renderParameters" class has methods for calculating all our matrices. Let's have a look at some of them.

```
void RenderParameters::computeMatricesFromInputs(float deltaTime, byte movementKeys)
```

This method takes in "how much time has passed", and "what keys have been pressed", and updates the view and model matrices. It only deals with translations, as the rotation is managed in the arcball class. For now this is not called anywhere! Our last exercise for this tutorial will be implementing some callbacks to take input and call this function. If you read through it, you will see that it takes the up, right, and forward axis from the camera rotation matrix (as we want to move it using our local orientation) but uses global axis to move the model.

```
Matrix4 getModelMatrix();
Matrix4 getViewMatrix();
Matrix4 getProjectionMatrix(float window_w, float window_h);
```

These do what you would expect. Now, there are a few things worth mentioning. The Model matrix is straightforward. Rotate first (around itself), then translate using the global XYZ. What about the View matrix? Well, it is simply the negative translation combined with the inverse rotation (which we can get with the transpose, as it's orthonormal).

Now our projection matrix is a bit more interesting. We are using the matrix described in the slides (Projective Pipeline) which points at +z, right handed, and defines normalized device coordinates from (-1,-1,0) to (1,1,1). Why is this relevant? OpenGL has some default behavior in some parts of its fixed pipeline that I am purposefully not conforming to for educational purposes (sorry). By default, it expects left handedness pointing at -z, and a NDCS [-1,1] on Z. This will have an impact on our next exercise.

## Exercise #9: Rendering

This is the moment you have been waiting for. You will finally see something rendered on your screen, and you will be able to tell if your code works!

So we need to do a few things. First, set general openGL properties related to the rendering we will be doing.

```
glClearColor(0, 0, 0, 0.0f);
glEnable(GL_FRAMEBUFFER_SRGB);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
glEnable(GL_CULL_FACE);
```

We want a black background (we will be raytracing in an empty space), so we set that as the clear colour. We will be calculating colours linearly on our shaders, but we want OpenGL to perform gamma correction (which we will be doing in our raytracer), so we enable GL_FRAMEBUFFER_SRGB to get that behavior. We want to enable the depth test, keeping the triangles that have the smaller depth (closest to the camera, remember we are defining 0-1 in our NDCS). We also want to enable backface culling. We can't however, forget to:

```
glFrontFace(GL_CW);
glClipControl(GL_LOWER_LEFT, GL_ZERO_TO_ONE);
```

Given that our matrix has different handedness. And since we are at it, also change the mapping from clipping space to screen space to know our z is mapped from 0-1 and not the default -1 to 1.

Now, let's write the rendering loop.

```
  // For speed computation
  double lastTime = glfwGetTime();
  int nbFrames = 0;

  do {
    // Measure speed
```

```
      double currentTime = glfwGetTime();
      float deltaTime = float(currentTime - lastTime);
      nbFrames++;
      if (deltaTime >= 1.0) { // If last prinf() was more than 1sec ago
         // printf and reset
         printf("%f ms/frame\n", 1000.0 / double(nbFrames));
         nbFrames = 0;
         lastTime += 1.0;
      }
      // Clear the screen
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
      Matrix4 ModelMatrix = renderParameters.getModelMatrix();
      Matrix4 ViewMatrix = renderParameters.getViewMatrix();
      Matrix4 ProjectionMatrix =
renderParameters.getProjectionMatrix(float(window_width)/2.0f,float(window_height));
      Matrix4 MVP = ProjectionMatrix * (ViewMatrix * ModelMatrix);

      //will add code here
   } // Check if the ESC key was pressed or the window was closed
   while (glfwGetKey(window, GLFW_KEY_ESCAPE) != GLFW_PRESS &&
      glfwWindowShouldClose(window) == 0);
```

So this is the overall loop. The code we will be writing now for the rendering will go inside this loop that runs while the esc key has not been pressed, or if the window has not been closed in another way.

First thing we do, is calculate our framerate. If one second has passed, we print how many frames have we rendered. That's simply done by accumulating number of frames, and counting the time until one second has passed. Next we clear the buffer, so we are not drawing on top of the previous frame, or using its depth values this time around. Then, we just grab our matrices from the render parameters.

```
   // Get a handle for our uniforms
   glUseProgram(programID);
   GLuint MatrixID = glGetUniformLocation(programID, "MVP");
   GLuint ViewMatrixID = glGetUniformLocation(programID, "V");
   GLuint ModelMatrixID = glGetUniformLocation(programID, "M");
   std::vector<GLuint> PosLightIDs;
   std::vector<GLuint> ColLightIDs;
   //Set the light position and color
   int currentLight = 0;
   for (Light* l : renderParameters.lights) {
      PosLightIDs.push_back(glGetUniformLocation(programID,
         ("lights[" + std::to_string(currentLight) + "].position").c_str()));
      ColLightIDs.push_back(glGetUniformLocation(programID,
         ("lights[" + std::to_string(currentLight) + "].color").c_str()));
      currentLight++;
   }
   GLuint nLightsID = glGetUniformLocation(programID, "nLights");
   GLuint emissiveID = glGetUniformLocation(programID, "meshMaterial.emissiveColor");
   GLuint diffuseID = glGetUniformLocation(programID, "meshMaterial.diffuseColor");
   GLuint ambientID = glGetUniformLocation(programID, "meshMaterial.ambientColor");
   GLuint specularID = glGetUniformLocation(programID, "meshMaterial.specularColor");
   GLuint shininessID = glGetUniformLocation(programID, "meshMaterial.shininess");
```

Remember how we declared uniforms in our shader? We now need to query their locations so we can set the actual values. glGetUniformLocation allows you to find just that given a program and the name of the parameter. This is something that can be done outside of the main loop if you are not expecting the shaders to change. We are going to do just that, and paste it before the "do" call in our main.

```
//First pass: Rasterisation
glUseProgram(programID);
// Send our transformation to the currently bound shader,
glUniformMatrix4fv(MatrixID, 1, GL_FALSE, MVP.columnMajor().coordinates);
glUniformMatrix4fv(ModelMatrixID, 1, GL_FALSE, ModelMatrix.columnMajor().coordinates);
glUniformMatrix4fv(ViewMatrixID, 1, GL_FALSE, ViewMatrix.columnMajor().coordinates);
//Set the light position and color
int cl = 0;
for (Light* l : renderParameters.lights) {
    Homogeneous4 lp = l->GetPositionCenter();
    Homogeneous4 c = l->GetColor();
    glUniform3f(PosLightIDs[cl], lp.x, lp.y, lp.z);
    glUniform3f(ColLightIDs[cl], c.x, c.y, c.z);
    cl++;
}
glUniform1i(nLightsID,currentLight);
```

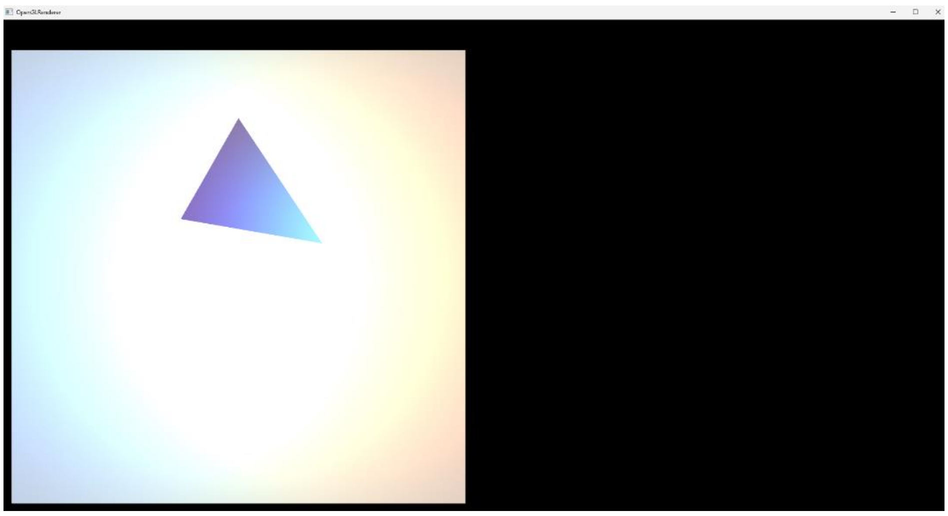
Now, we say which program we are going to use, and connect our variables from the c++ side to GLSL. (This call should come before all the other calls that use programID, including getting locations, that's why we put it up there earlier!) We use the locations we have queried in the previous block, then use "glUniform…" to set its value. Here, we are setting the matrix using glUniformMatrix4fv. Each type of parameter we are linking to GLSL will have a different appropriate function (floats, ints, etc.). 3f for our vectors of three floats, and 1i for our 1 integer.

```
//This will draw on the left side;
glViewport(0, 0, GLsizei(window_width / 2.0f), window_height);
for (int i = 0; i < vaoIDS.size(); i++) {
    //Setting material properties
    Cartesian3 d = objects[i].material->diffuse;
    Cartesian3 a = objects[i].material->ambient;
    Cartesian3 s = objects[i].material->specular;
    Cartesian3 e = objects[i].material->emissive;
    float shin = objects[i].material->shininess;
    glUniform3f(diffuseID, d.x,d.y,d.z);
    glUniform3f(ambientID, a.x,a.y,a.z);
    glUniform3f(specularID, s.x, s.y, s.z);
    glUniform3f(emissiveID, e.x, e.y, e.z);
    glUniform1f(shininessID, shin);
    glBindVertexArray(vaoIDS[i]);
    //Draw the triangles !
    glDrawArrays(GL_TRIANGLES,0,counts[i]);
}
```

```
// Swap buffers
glfwSwapBuffers(window);
glfwPollEvents();
```
Finally! We add our draw calls. First of all, a quick call to glViewport will tell OpenGL that we are only drawing on the left half of our screen, as the right half will be for the raytracing. If you paid attention, when we asked for a projection matrix we also said our window width was also just half. If you want things to render to the whole window, ignore this line, and also change the projection matrix.

For each textured object we have, we must set the material properties (using the locations we queried), then we can just bind the vaoID, and call drawArrays, saying we want to draw triangles, and how many of the vertices we have in our vao.

After rendering, we call swapBuffers so glfw shows what we just rendered to the screen, and process events that may have happened in the meantime. We can probably close our main and run things to finally see how it looks. If you pass the parameters "objects/triangle_backplane.obj objects/triangle_backplane.mtl" you should be able to see the following:



## 6) RAYTRACING SETUP

Finally, let's talk about raytracing. Most of your work in this assignment will be done in the "Raytracer" class. What it does is writing to an image the result of your calculations. Open Raytracer.cpp and .h to get an idea of what you will be implementing, so we can finish our setup.

The class has a public method called "Raytrace", which launches a separate thread running the method RaytraceThread, which currently does nothing. It has a "RGBAImage" called frame buffer, which has a comment saying `"//Output of your raytracer goes here".` Convenient, isn't it? Let's write some code that has a "dummy raytracing loop" and that writes some debug information on the image, so we can hook it up with our OpenGL Renderer.

### EXERCISE #10: RAYTRACING LOOP.

Navigate to your RaytraceThread function, and let's start with a simple loop.

```
for(int j = 0; j < frameBuffer.height; j++){
        for(int i = 0; i < frameBuffer.width; i++){
            Homogeneous4 color(i/float(frameBuffer.width), j/float(frameBuffer.height), 0);
            frameBuffer[j][i] = RGBAValue(
                                  linear_to_srgb(color.x),
                                    linear_to_srgb(color.y),
                                    linear_to_srgb(color.z),255);
            }
    }
```
Our future raytracer will be calculating one colour per pixel, across our whole framebuffer. Its dimensions are accessible in the ".width" and ".height" properties, which are correctly managed in the Resize method. Let's set the color to where in the framebuffer we are initially. Black would be 0,0, and yellow (red + green) would be 1,1. We use Homogeneous4 (our 4D vector type), a 0 to 1 floating point value so we have precise results.

However, framebuffer used 8bits per pixel, as in just 255 levels. Remember our lecture about colour spaces and SRGB? This is when it becomes relevant. The 2 conversion functions provided are implemented according to the standard so we are in the same colour space as our rasterization output. Let's use linear_to_srgb to gamma correct our pixel values and avoid all the issues of banding, and perception that we discussed.

There are two more things that we can add to this loop, first is making this computation parallel. As each pixel is calculated independently, let's use openmp to parallelize our outer loop. Add the following line before the for statement, and it should do it!

```
#pragma omp parallel for schedule(dynamic)
```
Oh, speaking of parallel, let's have some system of stopping our raytracer from running if we need to. The method stopRaytracer() has been conveniently implemented in a naïve way. It sets a Boolean variable "restartRaytrace" to true, and waits for it to be set to false to return successfully.

Let's just use it.

Add the following to the inside of your loop, this will set the variable to false, and return our raytracer:

```
if (restartRaytrace) {
        raytracingRunning = false;
        return;
    }
```
If you try to compile now, you will get the following error: "'return': jump out of OpenMP structured block not allowed". That means that if we tell OpenMP to parallelize a for loop, we cannot exit it until it is completed. The workaround here is simple. Let's move the #pragma statement to just before the for loop that changes i, instead of j. That will parallelize the calculation of groups of pixels in a row, instead of parallelizing calculation of full rows. Now all we need to do is place our return block out of that loop, and it should work.

Just remember to say raytracing is finished at the end of the function before returning.

```
raytracingRunning = false;
```
Let's now just instantiate our raytracer in our main function

```
#include "Raytracer.h"
Raytracer* myRaytracer;
```
Can be declared at the file level, as we will need access it in some callback functions that we will define later. (such as press a button to start raytracing!). Inside your main, after your call to loadModelGL you can paste the following:

```
myRaytracer = new Raytracer(&objects, &renderParameters);
myRaytracer->resize(int(window_width / 2.0f), window_height);
```
This starts it with our objects, and the target window size for our raytracer, which is full height, but half width, like our rasteriser. And that's it for our basic setup! Now we need to make sure our framebuffer is displayed on

the right side of your window. For that to work, you guessed it right, we need more shaders, uniforms, buffers, and things to set up OpenGL.

## Exercise #11: The death of glDrawPixels

If we were doing this in the "olden days", there used to be a function called glDrawPixels that simply grabbed a buffer and slapped it on the screen for you. It however had performance issues and relied on now deprecated behavior. As the overlords like things fast, they deprecated it, and now you must follow this tutorial to do it.

First, let's take care of the obvious part. We need to create an OpenGL texture where we can put our RGBAImage. This is conveniently a good opportunity for me to explain you how to use textures in OpenGL, as the objects we are rendering don't have them. Let's create a function to do this:

```
void loadScreenspaceTexture() {
    glGenTextures(1, &RaytracerTextureID);
```

Similarly to our arrays and buffers, we have an OpenGL call to generate an identifier for a texture for us. Go ahead and declare it in your main. This could be a parameter in the function, but realistically we will only have this one anyway, so let's keep it simple and place it at file level.

```
GLuint RaytracerTextureID;
```

Right, next we need to set the texture up so it is expecting the correct size, format, and type of data from our RGBAImage.

```
glBindTexture(GL_TEXTURE_2D, RaytracerTextureID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

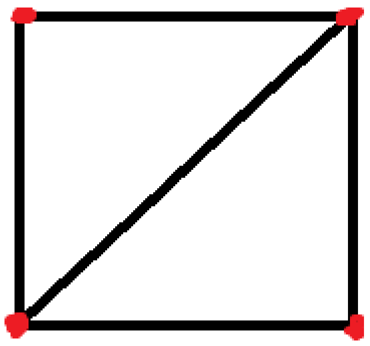
First, let's say that all following operations will be done to this specific texture we have created by binding it. We say the current "Texture_2D" that we are working on is the one we just created. We want the pixels from our texture to be mapped straight to our screen with no filtering, so we set both the MIN and MAG filter to nearest.

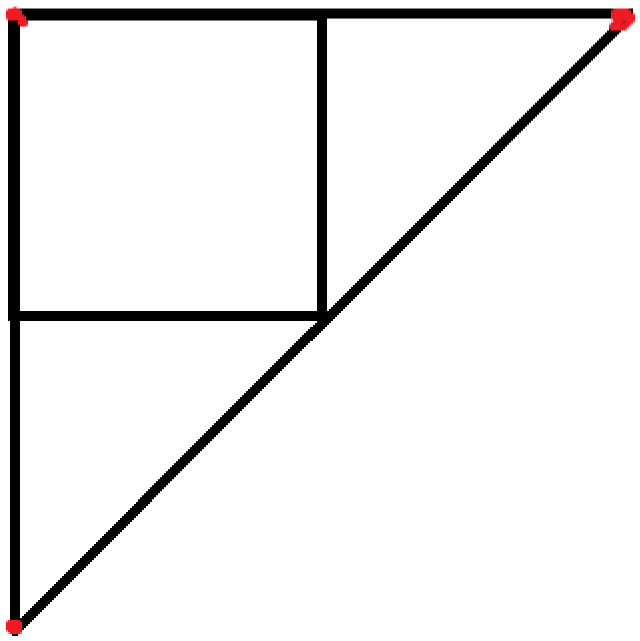
```
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, GLsizei(window_width / 2.0f), window_height, 0,
GL_RGBA, GL_UNSIGNED_BYTE, myRaytracer->frameBuffer.block);
```

Following that, we hand over the data to OpenGL saying we are putting data in the mip level 0, with a target format of SRGB (remember we did the linear_to_srgb conversion), dimensions and 0 sized border, and our source format is RGBA as well (that's how RGBAImage has it) and the pointer to the data.

If this was a normal texture that we load once and then use, at this point we could safely delete the buffer, as the data has been transferred to OpenGL. In our case, the raytracer will be updating the values of our texture, so we will repeat this call again when we are writing the rendering code. To finish it off, we unbind the current texture. In general, this is good practice. You just bind resources when you are going to use them.

```
glBindTexture(GL_TEXTURE_2D, -1);
```
Right, we got a texture. We need something to display it on. We have two options that easily come to mind.





Left: the obvious option, a fullscreen quad made of two triangles, with the corners mapped to the extremes of our NDCS coordinates, (-1-1), (-1,1), (1,1), and (1,-1). Then we just need to not transform them in the vertex shader. We can easily assign texture coordinates of 0 and 1 to them, and our texture would be mapped correctly.

Right: smarter way. We create a single large triangle, with corners at (0,0) (2.0) and (0,2) (y goes down in this diagram). This means we only process half the vertices, half the triangles. Moreover, the diagonal of our naïve approach will generate overlapping fragments for both the triangles, which is another waste of time!

Let's create a VAO that we use to call our renderer. The following should go in your main:

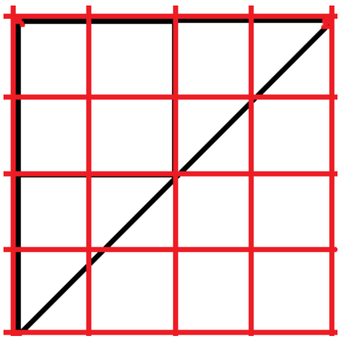
```
GLuint RaytracerVAO;
glGenVertexArrays(1, &RaytracerVAO);
glBindVertexArray(RaytracerVAO);
```
Do we need to create some vertices and UVS as we did before? We could, but given that we are being smart, let's continue this trend. Because we know exactly the position we want the vertices to be in the end, and their uvs, we just need to tell OpenGL to draw 3 vertices as a triangle, and "something" will be sent to our vertex shader, even if we have no buffers. Trust me, this is less weird than it sounds. Lets create two shaders: screenspace.vert and screenspace.frag.

```
#version 420
out vec2 v2fTexCoord;
void main()
{
```
Well, if we did not set attributes for our VAO, it means that we don't have any "in" parameters. But we still get something that we can work with. The vertex IDS! Our goal, looking at the diagram, is that for one vertex to have UVs of (0,0), and the other two to have (2,0) and (0,2), so the visible part of the screen maps our texture perfectly.

```
v2fTexCoord = vec2(
            gl_VertexID & 2,
            (gl_VertexID << 1) & 2
        );
```
Our IDs will be 0, 1 and 2. More precisely: 00,01,10 in binary! For vertexID 0, we got (0,0) as the & operations will always return 00. Now let’s look at vertexID 1.

```
01 & 10 = 00 = 0
01 << 1 = 10, 10 & 10 = 10 = 2
```
Tada! We have (0,2). If you run this now with vertexID 2, you will see that the output is (2,0). Perfect, now we just need our positions.

```
        gl_Position = vec4( 2.0 * v2fTexCoord - 1.0, 0.0, 1.0 );
    }
```
This gives us (-1,-1,0,1), (3,0,0,1), (0,3,0,1). Why are these the correct coordinates? Let’s draw a grid on top of our previous diagram:



If our top left corner is (-1.-1,0,1), count the units until the corner of the triangles, and you will see how we did the right maths! Likewise, our texture will be exactly mapping to our NDCS range.

Our fragment shader has no tricks:

```
#version 420

in vec2 v2fTexCoord;
out vec3 oColor;
layout(binding = 0) uniform sampler2D raytracerImage;
void main()
{
    oColor = texture( raytracerImage, v2fTexCoord ).rgb;
}
```
The new thing in this shader is using textures, which from the point of view of GLSL is pretty simple.  In GLSL, you declare a “sampler” of the type of texture you are going to be using, and it has the methods to read things from it. It knows what kind of filtering you are using, what should happen at the edges, and anything else you have set when loading your texture. We have a 2D texture, so that is a sampler2D. the “layout” qualifier in our uniform is something that we can do after version 4.2, which will be helpful when connecting it to the C++ code.

With our shaders finally written, we can use the functions we wrote on exercise 8 to compile them:

```
GLuint ssProgramID;
LoadShaders(ssProgramID, "screenspace.vert", "screenspace.frag");
loadScreenspaceTexture();
```
We should have everything. Let’s add the draw call after the loop that does the rasterization.

```
glUseProgram(ssProgramID);
glViewport(GLint(window_width / 2.0f), 0, GLsizei(window_width / 2.0f), window_height);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, RaytracerTextureID);
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB,GLsizei(window_width / 2.0), window_height, 0,
            GL_RGBA, GL_UNSIGNED_BYTE, myRaytracer->frameBuffer.block);
glBindVertexArray(RaytracerVAO);
glDrawArrays(GL_TRIANGLES, 0, 3);
```
Most of these calls should be familiar to you at this point, but let's quickly go through them as some things are slightly different. We choose the program we are going to use and set the viewport size and origin (notice we start where the previous one ends). Then we need to set our uniform, which in this case is only the texture.

We could do it in a similar way to what we had done for our other uniforms by getting their location from the name. But since OpenGL 4.2, we are also allowed to use that syntax of "location(binding=N)" to say which texture we are using. Your OpenGL implementation will allow up to a certain number of textures in a program execution (generally at least 16). These are called "texture units", and we need to say which one we are going to be using, and bind a texture to it. Since in our shader we said the binding was 0, we set the active texture to GL_TEXTURE0 before binding. We can simply add to GL_TEXTURE0 to get to the other ones in the case we are in a loop (GL_TEXTURE0 + i). From OpenGL 4.3, you can do layout(location=N) for all your uniforms, and then use N on c++ instead of doing getUniformLocation by name. Makes your code a bit smaller.

Given that our texture is updating every frame as the raytracing is running, we need to call glTexImage2D in the same way we did when creating the texture to update the data on the OpenGL side. Finally we bind the VAO, and say we are drawing one triangle, 3 vertices.

If you run it, you should see….. Nothing! If you go over to the "resize" method on your raytracer and add the following line:

```
frameBuffer.clear(RGBAValue(125.0f, 125.0f, 125.0f, 255.0f));
```
It will clear to gray instead of black. So if you run, at least you should be able to see the right half of your screen all gray. Please remove it before continuing as it was just a test!

## 7) FINAL TOUCHES

We are almost there, I promise. We need to be able to tell our raytracer to "go" and allow the user to place the camera. Given that all the other methods are implemented, that means we need to write in some code for user interaction. Next, will be cleanup (as good programmers we are) and you are good to go. Don't go for a break now, you are almost there.

### EXERCISE #12: CALLBACKS.

Thankfully doing this with GLFW is pretty straightforward. You tell it what function you will call when something happens, and then implement the function. Let's start with the easier one. We are writing all of these inside our initializeGL.

```
glfwSetKeyCallback(window, key_callback);
```
And declare our function. For each one of them there will be a specific function signature that you should use. For the key, is the following:

```
void key_callback(GLFWwindow* window, int key, int scancode, int action, int mods)
{
```

Given that this is a callback, it's not running in our main loop. It's normally good practice to just handle the input in a function that does that, and not have complex function calls that may stall execution of the program. We will use a Boolean to let our main loop know we want to raytrace.

```
//declare this at file level
bool launchRaytracer;
```

And from our parameters key and action, we can tell which key has triggered the event, and what happened. I'm checking when the key was released, as this guarantees it only runs it once. Press may trigger the event more than once.

```
//Raytracing keys
if (key == GLFW_KEY_R && action == GLFW_RELEASE) {
    launchRaytracer = true;
    renderParameters.printSettings();
}
```

That should be it. Let's also add a few events to change our render parameters. I will just now put the ones necessary for your first assignment.

```
if (key == GLFW_KEY_1 && action == GLFW_RELEASE) {
    renderParameters.interpolationRendering = !renderParameters.interpolationRendering;
    renderParameters.printSettings();
}
if (key == GLFW_KEY_2 && action == GLFW_RELEASE) {
    renderParameters.phongEnabled = !renderParameters.phongEnabled;
    renderParameters.printSettings();
}
if (key == GLFW_KEY_3 && action == GLFW_RELEASE) {
    renderParameters.shadowsEnabled = !renderParameters.shadowsEnabled;
    renderParameters.printSettings();
}
if (key == GLFW_KEY_4 && action == GLFW_RELEASE) {
    renderParameters.reflectionEnabled = !renderParameters.reflectionEnabled;
    renderParameters.printSettings();
}
```

Pretty simple. Now for movement our goal is to use the function defined on renderParameters "computeMatricesFromInputs". If you see how it is implemented, it takes a byte (8 bits) and checks which bit is turned "on". There are 8: Right mouse, Left mouse, forward, back, right, left, up, down. Perfect fit for a byte to hold that information, and for me to have you practicing using std::byte. Declare another variable on your main.

```
std::byte movementKeys;
```

Renderparameters.h defines what bit is what, so we just need to set those in our movementKeys!

```
// Move forward
    if (key == GLFW_KEY_W){
        if(action == GLFW_PRESS)
            movementKeys |= byte{ BIT_FW };
        if(action == GLFW_RELEASE)
            movementKeys &= ~byte{ BIT_FW };
    }
    if (key == GLFW_KEY_S) {
        if (action == GLFW_PRESS)
            movementKeys |= byte{ BIT_BACK };
        if (action == GLFW_RELEASE)
            movementKeys &= ~byte{ BIT_BACK };
    }
    if (key == GLFW_KEY_D) {
        if (action == GLFW_PRESS)
            movementKeys |= byte{ BIT_RIGHT };
        if (action == GLFW_RELEASE)
            movementKeys &= ~byte{ BIT_RIGHT };
    }
    if (key == GLFW_KEY_A) {
        if (action == GLFW_PRESS)
            movementKeys |= byte{BIT_LEFT };
        if (action == GLFW_RELEASE)
            movementKeys &= ~byte{ BIT_LEFT };
    }
    if (key == GLFW_KEY_Q) {
        if (action == GLFW_PRESS)
            movementKeys |= byte{ BIT_UP };
        if (action == GLFW_RELEASE)
            movementKeys &= ~byte{ BIT_UP};
    }
    if (key == GLFW_KEY_E) {
        if (action == GLFW_PRESS)
            movementKeys |= byte{BIT_DOWN };
        if (action == GLFW_RELEASE)
            movementKeys &= ~byte{ BIT_DOWN };
    }
}
```

It’s all the same. We check which key, and if it has been pressed or released. If it’s pressed, we use the Logical OR with our “mask” to set a certain bit up. If it’s released, we use the logical AND with the complement of our mask to turn it down. Let’s look at an example.

```
0b0000100   //currently pressing right,
0b0000001 | //logical or with byte(1), BIT_UP
---------
0b0000101 // now both are pressed!
0b1111110 & //logical or with ~byte(1), which has all bits flipped.
---------
0b0000100 // It's off now!
```

Great. Let’s move to our mouse controls. The following should go on your initializeGL.

```
glfwSetCursorPosCallback(window, mousePos_callback);
glfwSetMouseButtonCallback(window, mouse_button_callback);
```

Let’s start with the button. We will be setting movementkeys for the mouse too as they are used with the keyboard. Right button says we are controlling the model, and left button says we are controlling the camera.

Also, we will be using our Arcball class to implement rotations. RenderParameters has one for the camera and one for the model. It is expecting values in the range of [-1,1], and GLFW is giving us values in the pixel domain, so let’s start by converting those.

```
void mouse_button_callback(GLFWwindow* window, int button, int action, int mods)
{
    float scaledX = float(2.0f * mouseXpos - window_width) / float(window_width);
    float scaledY = float(window_height - 2.0f * mouseYpos) / float(window_height);
```

The next steps are fairly similar to what we did with the keyboard. Pressing the buttons will tell our renderparameters class what are we moving (as there are interactions with the keyboard, we only move while clicking!), but for rotating the camera we call the appropriate methods to drag the arcball.

```
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_PRESS)
    {
        renderParameters.CameraArcball.BeginDrag(scaledX,scaledY);
        movementKeys |= byte{ BIT_LEFTMOUSE };
    }
    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_PRESS)
    {
        renderParameters.ModelArcball.BeginDrag(scaledX, scaledY);
        movementKeys |= byte{ BIT_RIGHTMOUSE };
    }
    if (button == GLFW_MOUSE_BUTTON_LEFT && action == GLFW_RELEASE)
    {
        renderParameters.CameraArcball.EndDrag(scaledX, scaledY);
        movementKeys &= ~byte{ BIT_LEFTMOUSE };
    }
    if (button == GLFW_MOUSE_BUTTON_RIGHT && action == GLFW_RELEASE)
    {
        renderParameters.ModelArcball.EndDrag(scaledX, scaledY);
        movementKeys &= ~byte{ BIT_RIGHTMOUSE };
    }
}
```

This however, just starts and finishes the drag, we need some more code to do the actual dragging. Let’s implement the mouse movement callback. Did you notice that we were using undeclared variables? I did not. This goes declared at the file level.

```
double mouseXpos;
double mouseYpos;
```

And the function that updates it

```
void mousePos_callback(GLFWwindow* window,double x, double y) {
    // Get mouse position
    mouseXpos = x;
    mouseYpos = y;
    float scaledX = float(2.0f * mouseXpos - window_width) / float(window_width);
    float scaledY = float(window_height - 2.0f * mouseYpos) / float(window_height);
    if ((movementKeys & byte{ BIT_LEFTMOUSE }) != byte{ 0 })
        renderParameters.CameraArcball.ContinueDrag(scaledX,scaledY);
    if ((movementKeys & byte{ BIT_RIGHTMOUSE }) != byte{ 0 })
```

```
        renderParameters.ModelArcball.ContinueDrag(scaledX, scaledY);
}
```
Straightforward here. We save the mouse positions, as we need them for the mousedown function. Now we only need to check the movementkeys to see if which arcball we should be dragging. By doing the logic and (&), the result can only be different than 0, if that bit is on in our movementkeys. ContinueDrag then takes care of the rest for us.

Now, let's use some of the things we have defined here in our main loop. Starting with launching our raytracer. Put the following inside the rendering loop:

```
if (launchRaytracer) {
    myRaytracer->Raytrace();
    launchRaytracer = false;
}
```
Then, add the following before you set the matrices up in your shader.

```
renderParameters.computeMatricesFromInputs(deltaTime,movementKeys);
```
That's it! If you run the application now you should be able to move the camera with your mouse.

## EXERCISE #8 : CLEAN-UP

To wrap things up, let's add some clean-up code that ties in some loose ends. Let's start with textures and our programs, as we just have a few of those, then we delete our buffers and vaos in a short loop. We call this in our main together with cleaning up glfw, before we return.

```
glDeleteShader(programID);
glDeleteShader(ssProgramID);
glDeleteTextures(1, &RaytracerTextureID);
for (auto vaoID : vaoIDS) glDeleteVertexArrays(1,&vaoID);
for (auto vbID : vbIDS) glDeleteBuffers(1, &vbID);
for (auto nbID : nbIDS) glDeleteBuffers(1, &nbID);
for (auto tbID : tbIDS) glDeleteBuffers(1, &tbID);

myRaytracer->stopRaytracer();
glfwTerminate();
```
This will close the window and OpenGL.

There is still one case that we would like to do, which is make the window resizeable! Let's set the callback for a resize in our initializeGLFW,  and the implementation of the function.

```
glfwSetWindowSizeCallback(window, window_resized); //put this inside initializeGLFW()
void window_resized(GLFWwindow* window, int width, int height) {
    window_width = width;
    window_height = height;
    myRaytracer->stopRaytracer();
    myRaytracer->resize(int(width / 2.0f), height);
}
```

That's it! This tutorial is complete. You have a basic openGL renderer that can be used to complete the raytracing part of your first assignment.