
TUTORIAL #1 RAYTRACING:

Goal

In this tutorial you will be guided through completing the basic code for the Raytracing component of your assignment. Basic classes of the QT framework will be introduced, auxiliary classes to handle the scenes and basic rendering will be implemented, and your outermost loop of the Raytracing algorithm, together with skeletons of functions that should be implemented as part of the assessed component.

GETTING STARTED

University Linux: module load legacy-eng module load dependent-modules module add gcc module add qt/5.15.2 qtcreeator RaytraceRenderWindow.pro	MacOSX / Windows Install QtCreator Double click RaytraceRenderWindow.pro to open in QtCreator
---	---

- Select the platform to compile to (32 or 64 bits). For windows MinGW tends to work better than MSVC for being compatible with the linux codebase.
- Click details to select the build folder
- Click Configure Project

And your project has been loaded. On the left you can see all of the files that belong to your project. We will go through a few different components of them.

0) MAIN

As you well know (I expect) the entry point for a C++ application is the main function. So please open this file so we can inspect it. It is commented in a fair amount of detail. So let's just highlight a few key parts of it.

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

```
int main(int argc, char **argv)
{ // main()
  // initialize QT
  QApplication renderApp(argc, 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. In our case, we are running a QApplication, so we need to first pass them forward to its constructor for initialization (more on QT later).

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 texture|material" << std::endl;
  // and Leave
  return 0;
} // bad arg count

```

Let's **run the application** to see it in action. If you click  the program will run without the debugger. The expected result is to see the error message you have added!

If you click  it will launch with the debugger. For that, make sure that you have the  configuration selected. This way, QT will include debug information with your program, and allow you to put breakpoints in certain lines, and inspect the values some variables have. Click left to the line number of where you wrote the if statement, run the program with the debugger, and then hover with your mouse on top of argc to see its value. Follow the execution of the program using the controls provided.



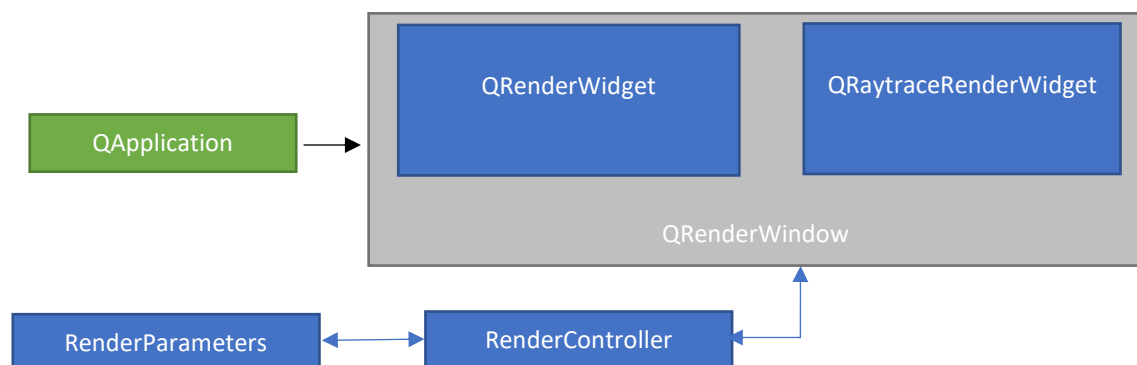
: Continue to next breakpoint, stop debugging, step over a function call, step inside a function call, and step out of a function call.

Now before we look at the rest of our main, let's understand the QT framework which will be the skeleton of our raytracer.

1) QT(CUTE/CUTIE/CUE TEE/QUEUE TEE/QUEUEGH TEUGH) CLASSES

If you never heard of it, this is QT, a cross platform software that the creators want really hard for you to call "cute", when in actual fact it would be a lot less forced if they wanted you to call it "cutie". It allows you to write cross platform applications with a gui. We will be using it to allow us to have some UI controls for our raytracer and displaying two rendering windows: one for a rasterization preview, and one for the raytracer's results.

Although teaching you QT is not one of my objectives in this tutorial, it's helpful if you understand what is going on with your code. But here is the basic diagram:



QAPPLICATION: What will control our main loop. It makes sure the correct functions are called when you click buttons (signals and slots on QT lingo), handles input and output, calls render code, etc.

Now we are using a “Model-View-Controller” paradigm for the GUI that controls our raytracer. The three main components of this are:

RENDERPARAMETERS: Our Model. Keeps all the updated parameters of our rendering system that need to be accessed by the rendering code. Open the .h and the .cpp file and look through the variables that are already defined. There are several Boolean values with different functions that we want to disable/enable in our Raytracer, and the transformations we can apply.

```
RenderParameters():
    xTranslate(0.0),
    yTranslate(0.0),
    zTranslate(0.0),
    interpolationRendering(false),
    phongEnabled(false),
    fresnelRendering(false),
    shadowsEnabled(false),
    reflectionEnabled(false),
    refractionEnabled(false),
    monteCarloEnabled(false),
    centreObject(false),
    orthoProjection(false)
{
    // because we are paranoid, we will initialise the matrices to the identity
    rotationMatrix.SetIdentity();
}
```

Here is the constructor where all the values are initialized. So if you ever want to change the “default” for any of the checkboxes, it’s here the place to do it.

Further down, there is a list of defines that define a lot of the ranges in the how the sliders are allowed to move:

```
// now define some macros for bounds on parameters
#define TRANSLATE_MIN -1.0f
#define TRANSLATE_MAX 1.0f

#define ZOOM_SCALE_LOG_MIN -2.0f
#define ZOOM_SCALE_LOG_MAX 2.0f
#define ZOOM_SCALE_MIN 0.01f
#define ZOOM_SCALE_MAX 100.0f

#define LIGHTING_MIN 0.0f
#define LIGHTING_MAX 1.0f

#define SPECULAR_EXPONENT_LOG_MIN -2.0f
#define SPECULAR_EXPONENT_LOG_MAX 2.0f
#define SPECULAR_EXPONENT_MIN 0.01f
#define SPECULAR_EXPONENT_MAX 100.0f

// this is to scale to/from integer values
#define PARAMETER_SCALING 100
```

These will be connected to our controller in a little bit.

RENDERCONTROLLER: Our Controller. Contains the functions that map the UI to values in our model. Open the .h and .cpp to see what it has. If this was a QT assignment, we would spend quite some time talking about this. It is not the case so let's keep it short.

Looking at the .h you will see it mostly has methods associated to events that we expect to happen to our interface (e.g. monteCarloBoxChanged). And that is its job! Whenever something in the window changes, it will access and update the variables in our RenderParameters. If you look at the constructor, you will see how QT is making sure these things happen. Example:

```
QObject::connect(renderWindow->monteCarloBox, SIGNAL(stateChanged(int)), this,
SLOT(monteCarloBoxChanged(int)));
```

This tells the QApplication to watch the object in our renderWindow called "monteCarloBox" (which is a checkbox that we will see in a bit), so when it emits a "SIGNAL" called "stateChanged" that has an int parameter. When this signal is emitted, pass it to this object, more specifically, call the function "monteCarloBoxChanged" with said int parameter so we can process it. This is what the "connect" function does.

```
void RenderController::monteCarloBoxChanged(int state)
{
    // reset the model's flag
    renderParameters->monteCarloEnabled = (state == Qt::Checked);
    // reset the interface
    renderWindow->ResetInterface();
}
```

And in the actual function, we update the value in our model. Done! Model View Controller in action.

RENDERWINDOW: The window that will have our UI. Our "View". It contains all the widgets for all the visual elements you see in the window, the two rendering widgets. Open the .h and .cpp to see what it does.

You will notice in the .h that it has a reference to the rendered objects and parameters right at the start. We will come back to why we need these when discussing the modelling and the rendering process. The rest of the variables are all the visual elements we see in our window. QCheckboxes, QSliders, QLabels, and our two rendering widgets "RenderWidget" and "RaytraceRenderWidget".

If you look through the .cpp file now, you will see what is the main type of behavior we are defining here.

```
monteCarloBox = new QCheckBox("Monte-Carlo", this);
```

We create a widget

```
windowLayout->addWidget(monteCarloBox,8,3,1,1);
```

Then place it somewhere within our "gridlayout". This puts it at row 8, column 3, and occupying 1 cell in rows and columns.

Finally, There are two special widgets here, which we will shortly discuss:

RENDERWIDGET, RAYTRACERENDERWIDGET: These two classes are implementations of "QOpenGLWidget", an abstract class. It mainly contains a window where rendered result is displayed, and methods related to doing the actual rendering. The size of this widget is defined by the layout in your renderWindow, so there are methods to run when it is resized since it may affect how we need to render. There are also methods to handle mouse input, as we want to allow people to click directly on the models More on this later on 4) and 5).

Open main.cpp again to see where all of these are created, and where the main loop of the application is started.

```
// set the initial size
renderWindow.resize(1600, 720);
// show the window
renderWindow.show();
// set QT running
return renderApp.exec();
```

Here you can adjust the initial size of the window.

EXERCISE #1

Let's create a button that calls a function that will do the raytracing on our widget. Start by adding a button to our RenderWindow.h:

```
//button for raytracing
QPushButton *raytraceButton;
```

Initialize it in the constructor on RenderWindow.cpp:

```
raytraceButton = new QPushButton("Raytrace", this);
```

Then place it within our gridLayout. Let's put it at the last column, and being the only button in that column. For that we will use the row span to be "nStacked", which says how many rows of widgets we have stacked between the two renderWidgets.

```
windowLayout->addWidget(raytraceButton, 0, 6, nStacked, 1);
```

That's it for the window. Now let's connect things in the controller. Like all the other widgets, let's "connect" things. First, let's declare a function to be called when we click that button. Add to the .h

```
void raytraceCalled();
```

Now we can add the following line to the constructor:

```
QObject::connect(renderWindow->raytraceButton,
    SIGNAL(released()), this, SLOT(raytraceCalled()));
```

Right, now let's define what happens when this button is called. We want all of our raytracing code to be handled by the RaytraceRenderWidget class. Given that it is owned by the RenderWindow class, let's call a method in the window to invoke the appropriate behavior in its widget. Add the following implementation to RenderController.cpp:

```
void RenderController::raytraceCalled()
{
    renderWindow->handle_raytrace();
}
```

And on your RenderWindow, define this method:

```
//add to .h file
void handle_raytrace();
//add to .cpp file
void RenderWindow::handle_raytrace()
{
    raytraceRenderWindow->Raytrace();
}
```

Finally, let's add the function for raytracing on the widget. Don't get too excited, let's just print something there.

```
//add to .h file
// routine that generates the image
void Raytrace();
//add to .cpp file
void RaytraceRenderWindow::Raytrace()
{
    std::cout << "One day I will raytrace." << std::endl;
}
```

Done! Everything is connected. So we should be able to see this print if we run the application, right? Yes, but we are still getting stuck with our print saying we don't have enough parameters. Let's look at the "modelling" in the next section, so we can provide our application with some parameters.

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 to see how the model is processed.

After our if from exercise #1, there is a standard block of code to open the file using an ifstream, and checking if the file exists, returning an error when it doesn’t. So if you write something that is not the right path, you will see the error message from this if.

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

After that, we are 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 after this we have everything read and accessible in our objects!

EXERCISE #2

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.

Add this vector to the class,

```
//add this to the beginning
#include <vector>
//add this to the class definition
std::vector<Light*> lights;
```

And now let's finish implementing the function that finds them. It is defined in 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 is 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);
}


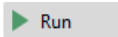
```

And done! This should set all the lights in our RenderParameters. Let's call it in our main after we declare our RenderParameters, and with a little print to see if it worked.

```

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

```

Oh. Yes. We never got to try our button too! Let's test both at the same time. Click  , then let's set the command line input parameters:

```

Command line arguments: ..\objects\cornell_box.obj ..\objects\cornell_box.mtl

```

Now run the application, and see the results of both our button, and the number of lights that are parsed from the file!

Finally, let's just remember to clean up our memory, and add a destructor to our RenderParameters, as we have created several lights. Add this to the .h file

```

~RenderParameters(){
    for (unsigned int i = 0;i<lights.size();i++) {
        delete(lights.at(i));
    }
}

```

Now it's nice and tidy.

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.

EXERCISE #3

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

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.

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.

```
Ray(Cartesian3 og, Cartesian3 dir);  
Cartesian3 origin;  
Cartesian3 direction;
```

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 BASIC RENDERING

Did you ever hear the tragedy of OpenGL immediate mode? I thought not. It’s not a story the Jedi, I mean, the modern programmers will tell you. It is a pathway to many abilities some consider to be... unnatural.

Anyway. For the sake of history (and some areas where it may still be useful) this section of the code uses OpenGL immediate mode to render, so not the stuff I will be teaching you. A few things are still the same, others not.

For now our window does not show anything. Let's look at RenderWidget.cpp to figure out what is missing to get our model rendered. The first relevant function we see is "initializeGL".

```
glShadeModel(GL_SMOOTH);
glEnable(GL_LIGHTING);
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_FALSE);
float globalAmbientColour[4] = {0.0f, 0.0f, 0.0f, 1.0f};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, globalAmbientColour);

// background is black!
glClearColor(0, 0, 0, 1.0);
```

This is a general setup of our rendering. We are saying we use lighting, with smooth shading (gouraud), and just shading one side of the polygons. We also turn off the default global ambient light that OpenGL has. Then finally, setting the color of our background to black!

Next up, resizeGL. Whenever our RenderWindow is resized, our widgets will also be resized, and that affects our rendering.

```
// reset the viewport
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
double aspectRatio = double(w) / double(h);
if (aspectRatio > 1.0)
    glOrtho(-aspectRatio, aspectRatio, -1.0, 1.0, 0.0, 2.0);
else
    glOrtho(-1.0, 1.0, -1.0/aspectRatio, 1.0/aspectRatio, 0, 2.0);
```

First we just adjust the size of our viewport, as the total size we have available to render has changed. Then, this also affects our projection, as the aspect ratio of the window has also probably changed!

We are using an orthographic projection here (rectangular), so we simply fix the smaller of the dimensions to be in the interval [-1,1], and correctly set the other one to not produce any distortion. And we are only going to see things in depths 0-2, due to our near and far planes.

Finally, actual rendering.

```
glEnable(GL_DEPTH_TEST);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

First we are saying "yes we will be doing the depth test", which you will learn about in a few lectures, then clearing our buffers so we can draw on them. This basically applies the color defined in glClearColor to the whole buffer.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(renderParameters->xTranslate, renderParameters->yTranslate,
renderParameters->zTranslate-1);
glMultMatrixf(renderParameters->rotationMatrix.columnMajor().coordinates);
```

Then, we set our matrices that represent our transformations. We tell OpenGL we will be editing the modelView matrix. We first set it to identity, then apply a translation following the values we got from the sliders on our

renderParameters (see how it's useful?). For the Z coordinate, we add -1 to it, as we want to push things back a little bit by default. Our camera is at zero, looking towards negative Z, so if a model is centered at 0 we wouldn't see it. Then, we apply our rotation matrix just by multiplying by what we get from the rotationMatrix in our renderParameters. OpenGL wants it in a columnMajor format, so we do that.

```
for (unsigned int i = 0; i < texturedObjects->size(); i++) {  
    texturedObjects->at(i).Render();  
}
```

Then we go through all of the objects we have, and call the "Render" method that is implemented for you. Right click it, and choose "follow symbol under cursor" to see what it does.

The first 45 lines are roughly just copying the material into float arrays, or giving them default values. Then we iterate the ThreeDModel. 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. So instead of just going "per face, render these three vertices", we have to render them as a "triangle fan". Thankfully, GL_TRIANGLE_FAN is a thing, so we just need to say that we will be providing our data as elements of a triangle fan, and it will work.

```
glBegin(GL_TRIANGLE_FAN);  
for (unsigned int faceVertex = 0; faceVertex < faceVertices[face].size(); faceVertex++)  
{  
    glNormal3f(  
        normals        [faceNormals    [face][faceVertex] ].x,  
        normals        [faceNormals    [face][faceVertex] ].y,  
        normals        [faceNormals    [face][faceVertex] ].z);  
    glTexCoord2f(  
        textureCoords  [faceTexCoords  [face][faceVertex] ].x,  
        textureCoords  [faceTexCoords  [face][faceVertex] ].y);  
    glVertex3f(  
        vertices       [faceVertices   [face][faceVertex]].x,  
        vertices       [faceVertices   [face][faceVertex]].y,  
        vertices       [faceVertices   [face][faceVertex]].z);  
} // per triangle  
glEnd();
```

And we are done. But why can't we see anything? Oh, yes. The lights are off!

EXERCISE #4

Let's use the information from the lights we have in our scene in openGL. Navigate to RenderWidget.cpp, and let's set up some lights.

The function we implemented on Exercise #2 initializes a vector of lights on RenderParameters. Very smart of us to do so. Now it makes our lives a lot simpler. So add the following code snippets before you call the rendering code. Let's start with a simple loop to go through all of our lights.

```
for (unsigned int i = 0; i < renderParameters->lights.size(); i++)  
{  
    glEnable(GL_LIGHT0+i);
```

This enables a light. On Immediate mode, you will have up to "GL_MAX_LIGHTS" lights defined in your scene. They are defined as GL_LIGHT0, GL_LIGHT1, and so on. As these are just some #define of an int, it happens that you can add to GL_LIGHT0 and it will end up being the correct define. Weird but okay.

```
Homogeneous4 p= renderParameters->lights[i]->GetPositionCenter();
float lightPosition[4] = {p.x,p.y,p.z,p.w};
Homogeneous4 c = renderParameters->lights[i]->GetColor();
float lightColor[4] = {c.x,c.y,c.z,c.w};
```

We need to set the position and color of our lights. So we just get those from the file and make them into float arrays. Now we are ready to pass them to OpenGL.

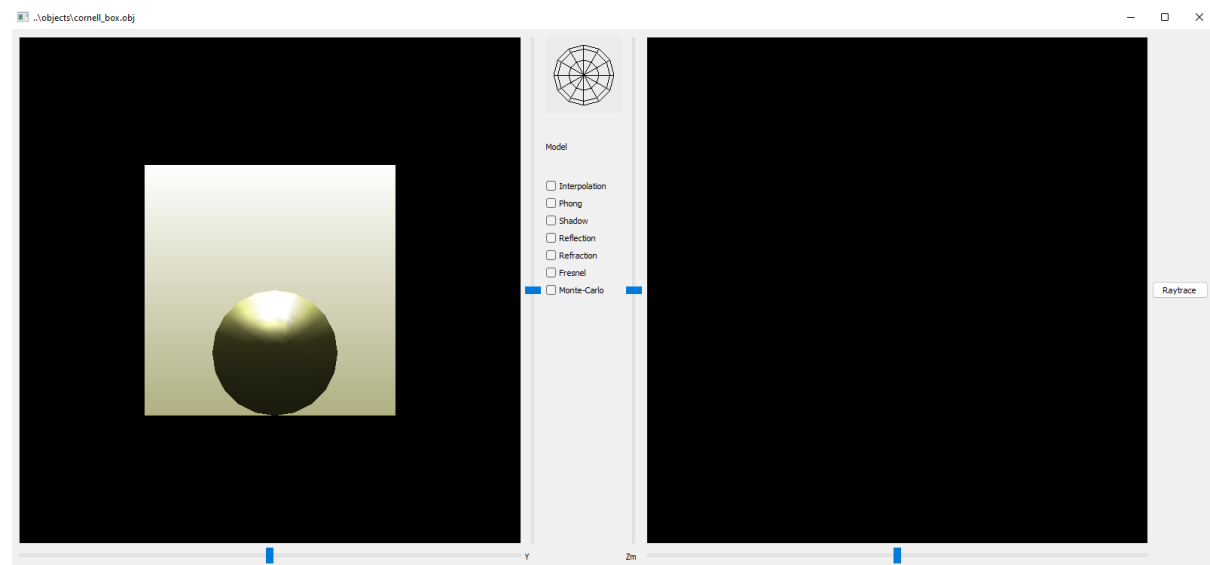
```
glLightfv(GL_LIGHT0+i, GL_POSITION, lightPosition);

glLightfv(GL_LIGHT0+i, GL_AMBIENT, lightColor);
glLightfv(GL_LIGHT0+i, GL_DIFFUSE, lightColor);
glLightfv(GL_LIGHT0+i, GL_SPECULAR, lightColor);

glLightf(GL_LIGHT0+i, GL_CONSTANT_ATTENUATION, 0.0f);
glLightf(GL_LIGHT0+i, GL_LINEAR_ATTENUATION, 0.0f);
glLightf(GL_LIGHT0+i, GL_QUADRATIC_ATTENUATION, 1.0f);
```

glLightfv sets a property for the light in question, for the property on the second parameter, with the value provided on the third. The “fv” version takes a float vector, and that is why we converted things. You probably noticed that we need to set a different color for ambient, diffuse and specular. Weird, but that’s just one way of modelling light interaction with surfaces.

We close the for loop with a curly bracket, and done! Let’s run the program with ..\objects\cornell_box.obj and ..\objects\cornell_box.mtl. You should see something similar to this.



Which is a bit underwhelming. Let’s add a version that does Perspective projection! Go to ResizeGL on RenderWidget.cpp, and replace our little if with ortho with:

```
if (aspectRatio > 1.0)
    if(renderParameters->orthoProjection)
        glOrtho(-aspectRatio, aspectRatio, -1.0, 1.0, 0.0, 2.0);
    else
        glFrustum (-aspectRatio * 0.01, aspectRatio * 0.01, -0.01, 0.01, 0.01, 200.0);
else
    if(renderParameters->orthoProjection)
```

```

        glOrtho(-1.0, 1.0, -1.0/aspectRatio, 1.0/aspectRatio, 0, 2.0);
    else
        glFrustum ( -0.01, 0.01, -0.01/aspectRatio, 0.01/aspectRatio, 0.01, 200.0);

```

glFrustum does a perspective projection. Try playing with the values in glFrustum to see its effects on how things look.

Now the last thing that is left is allowing us to change to the orthoProjection. After our tutorial with the Raytrace button, are you able to add it to your window? Try to do it without the following hints. If you get stuck, here is what you need to do and where

```

//RenderWindow.h
QCheckBox* orthographicBox;
//RenderWindow.cpp constructor
orthographicBox= new QCheckBox("Orthographic",this);
windowLayout->addWidget(monteCarloBox,9,3,1,1);
//RenderWindow.cpp resetInterface
orthographicBox->setChecked(renderParameters->orthoProjection);
orthographicBox->update();
//RenderController.cpp constructor
QObject::connect(renderWindow->orthographicBox,SIGNAL(stateChanged(int)),
                 this,SLOT(orthographicBoxChanged(int)));
//RenderController.cpp new method
void RenderController::orthographicBoxChanged(int state){
    renderParameters->orthoProjection = (state == Qt::Checked);
    renderWindow->ResetInterface();
}
//RenderController.h
void orthographicBoxChanged(int state);
//PaintGL() at RenderWidget.cpp
resizeGL(width(),height());

```

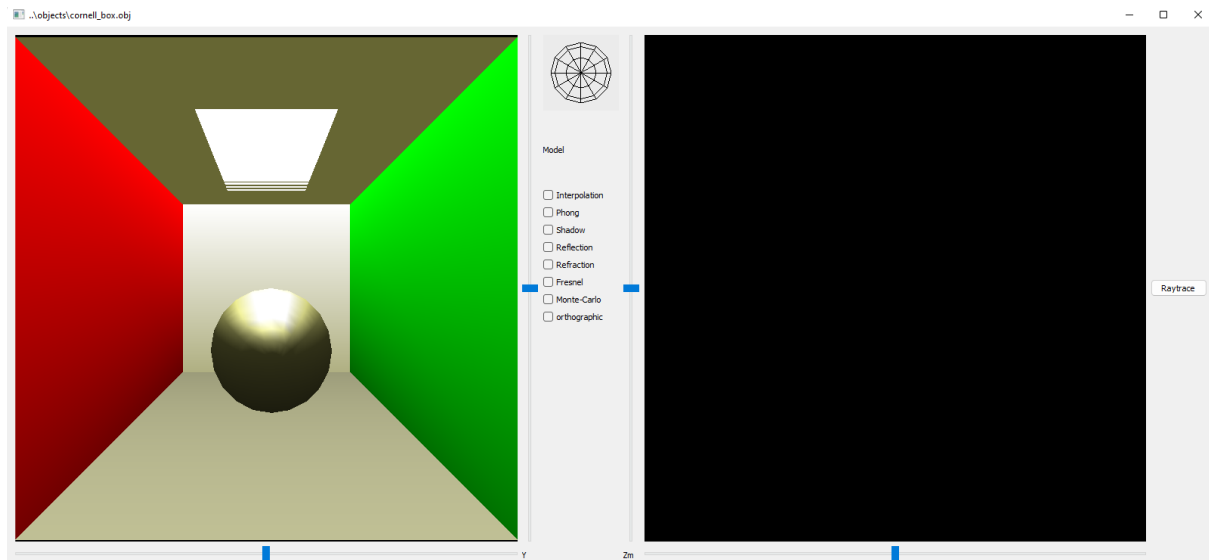
Try running it now and changing the checkbox. Did nothing happen? Try resizing the window. Oh. Yes. We forgot that we just set the projection when we resize the window. For a quick hack to get this sorted, let's do this:

```

//PaintGL() at RenderWidget.cpp
resizeGL(width(),height());

```

As bad as it looks, we know our resize function just calculates the projection again. If we were allocating buffers or stuff like that, probably not the best course of action. Let's run again with the cornell_box to see what's our result.



Ta-da! You have a perspective projection working. We are ready to move on to the raytracing!

5) RAYTRACING SETUP

Finally, let's talk about raytracing. Open RaytraceRenderWidget.cpp and go to the "Raytrace" method you have defined. Our goal is to run the raytracing algorithm inside this function. So we know that the basic loop is to go through each pixel, and calculate the color for it. Given that we expect each iteration to take a while, we will want to make it parallel, and we most likely also want to have some data structures to support our code. This is going to be a long exercise.

EXERCISE #5

Let's start with something simple, where we just set the color for the pixel.

```
frameBuffer.clear(rgbaValue(0.0f, 0.0f, 0.0f, 1.0f));
for(int j = 0; j < frameBuffer.height; j++){
    for(int i = 0; i < frameBuffer.width; i++){
        Homogeneous4 color(i/float(frameBuffer.height), j/float(frameBuffer.width), 0);
        frameBuffer[j][i] = rgbaValue(color.x*255.0f, color.y*255.0f, color.z*255.0f, 255.0f);
    }
}
```

We are calculating the colors as floats, and our framebuffer is an RGBImage that takes RGBValues. So at the end we just adjust the range to be 0-255 instead of 0-1. As debug, we are painting every pixel with its coordinates. Resize the window a little bit and run it a few times to see how it behaves.

Although it runs instantly now since we are not calculating any colours, let's add a little bit of code to pretend things are being done to see how our application behaves. Add the following to the for loop.

```
int k = 100000; while(k-->0);
```

The application hangs, and it takes a while! Let's add a little directive to allow this loop to run in parallel to see if it helps. Add the following line before the for loop.

```
#pragma omp parallel for schedule(dynamic)
```

And add the following to your .pro file

```
#adding openMP
```

```
QMAKE_CXXFLAGS+= -fopenmp -Wall
```

```
LIBS += -fopenmp
```

OpenMP allows you to quickly parallelize for loops in this way. This will use all of your available threads to parallelize that loop! My computer has cooling issues, so sometimes I need to limit the number of threads it uses. You can do that by calling the “omp_set_num_threads(int n)” function from <omp.h>. If you run this now, you will notice that you got the output a little bit faster. But if you add another zero, the application hangs again! And even worse, you only see the final results of your rendering at the end. What if something is wrong? We need to allow it to show us a preview of the process.

For that, let’s use another way of multi-threading. Let’s have a separate thread that is not in our main QT application loop to calculate the raytracing, and the QT thread will keep control of the mouse interaction and updating our screen. Let’s extract our raytracer to a separate function in this class, but let’s already add gamma correction to it as well:

```
//add to RaytraceRenderWidget.h
```

```
void RaytraceThread();
```

```
//add to RaytraceRenderWidget.cpp
```

```
void RaytraceRenderWidget::RaytraceThread()
```

```
{
    framebuffer.clear(rgbaValue(0.0f, 0.0f, 0.0f, 1.0f));

    #pragma omp parallel for schedule(dynamic)
    for(int j = 0; j < framebuffer.height; j++){
        for(int i = 0; i < framebuffer.width; i++){
            Homogeneous4 color(i/float(framebuffer.height), j/float(framebuffer.width), 0);
            int k = 100000; while(k-->0);
            //Gamma correction...
            float gamma = 2.2f;
            //We already calculate everything in float, so we just do gamma correction
            before putting it integer format.
            color.x = pow(color.x, 1/gamma);
            color.y = pow(color.y, 1/gamma);
            color.z = pow(color.z, 1/gamma);
            framebuffer[j][i] = rgbaValue(color.x*255.0f,
                                           color.y*255.0f, color.z*255.0f, 255.0f);
        }
    }
    std::cout << "Done!" << std::endl;
}
```

So now we want to launch a new thread to do this every time we click the button. So let’s change our Raytrace function to do that! First, let’s add the following to the .h

```
#include <thread>
```

```
std::thread raytracingThread;
```

Then change the body of “Raytrace” to the following.

```
raytracingThread = std::thread(&RaytraceRenderWidget::RaytraceThread, this);
raytracingThread.detach();
```


This launches a new thread with the function we just wrote, and detach from this main thread, so QT can continue to operate normally. Detach is what actually launches the thread! If we used the other option “join”, we would be waiting for it to complete, which is not what we want to do here.

If you run it now, you will notice that you are able to at least click the checkboxes. But you cannot see the window being painted. QT actually only redraws the screen when some event happens. We will need to force the screen to be repainted. So define the following function

```
//add to .h
void forceRepaint();
//add to .cpp
void RaytraceRenderWidget::forceRepaint(){
    update();
}
```

Now, let's set something up to call this every now and then. We can use QTimer for that.

```
#include <QTimer>
//add to the constructor
QTimer *timer = new QTimer(this);
connect(timer, &QTimer::timeout, this, &RaytraceRenderWidget::forceRepaint);
timer->start(30);
```

The default behaviour is good enough for us. When the timer expires, our function is called, then it's automatically restarted! If you run things again, you will see the screen being incrementally painted.

Given that our raytracing process will take a while and we just allowed people to be moving things around after we start the process, it would be good to have some data structure to hold a “snapshot” of our ThreeDModels in a given position so nothing changes while we are raytracing. While we are at it, let's just make this scene a little bit more useful overall for us. As we expect this representation to be made of Triangles, let's just create a Triangle class with some basic functions.

```
class Triangle
{
public:
    Homogeneous4 verts[3];
    Homogeneous4 normals[3];
    Homogeneous4 colors[3];
    Cartesian3 uvs[3];

    Material *shared_material;
    Triangle();
};
```

With just the constructor definition on the .cpp:

```
Triangle::Triangle()
{
    shared_material= nullptr;
}
```

We will add more functionality to it later. Now create a new class called “Scene”, and let's start with this definition:

```
class Scene
```

```

{
public:
    std::vector<ThreeDModel>* objects;
    RenderParameters* rp;
    std::vector<Triangle> triangles;
    Scene(std::vector<ThreeDModel> *texobjs, RenderParameters *renderp);
    void updateScene();
};

```

(At this point I'm expecting you to be able to add the necessary includes, so I'm skipping that)

So we want to initialize our scene with our models and parameters, and when we call the method "updateScene" we want to apply the transformations, and save the model to a list of triangles, where each one has all the necessary information for rendering. Let's start with the implementation then.

```

void Scene::updateScene()
{
    triangles.clear(); //Clear the list so it can be populated again
    for (int i = 0; i < int(objects->size()); i++)
    {
        typedef unsigned int uint;
        ThreeDModel obj = objects->at(uint(i));
        for (uint face = 0; face < obj.faceVertices.size(); face++)
        {
            for (uint triangle = 0; triangle < obj.faceVertices[face].size()-2; triangle++)
            {
                Triangle t;
                for (uint vertex = 0; vertex < 3; vertex++)
                {
                    uint faceVertex = 0;
                    if (vertex != 0)
                        faceVertex = triangle + vertex;

                    Homogeneous4 v = Homogeneous4(obj.vertices[obj.faceVertices[face][faceVertex]].x,
                                                    obj.vertices[obj.faceVertices[face][faceVertex]].y,
                                                    obj.vertices[obj.faceVertices[face][faceVertex]].z);

                    t.verts[vertex] = v;

                    Homogeneous4 n = Homogeneous4(obj.normals[obj.faceNormals[face][faceVertex]].x,
                                                    obj.normals[obj.faceNormals[face][faceVertex]].y,
                                                    obj.normals[obj.faceNormals[face][faceVertex]].z,
                                                    0.0f);

                    t.normals[vertex] = n;

                    Cartesian3 tex = Cartesian3(obj.textureCoords[obj.faceTexCoords[face][faceVertex]].x,

```

Again, we are getting models as "faces", and we are dealing 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.

```

//this is our vertex before any transformations. (world space)
Homogeneous4 v = Homogeneous4(obj.vertices[obj.faceVertices[face][faceVertex]].x,
                                obj.vertices[obj.faceVertices[face][faceVertex]].y,
                                obj.vertices[obj.faceVertices[face][faceVertex]].z);

t.verts[vertex] = v;

Homogeneous4 n = Homogeneous4(obj.normals[obj.faceNormals[face][faceVertex]].x,
                                obj.normals[obj.faceNormals[face][faceVertex]].y,
                                obj.normals[obj.faceNormals[face][faceVertex]].z,
                                0.0f);

t.normals[vertex] = n;

Cartesian3 tex = Cartesian3(obj.textureCoords[obj.faceTexCoords[face][faceVertex]].x,

```

```

        obj.textureCoords[obj.faceTexCoords[face][faceVertex]].y,
        0.0f);
t.uvs[vertex] = tex;
t.colors[vertex] = Cartesian3( 0.7f, 0.7f, 0.7f);

```

This assigns all of them to the triangle we created. Now let's close the for loop, assign the material, and add the triangle! Just in case the material has not been set up for a given model, let's create a "default material" for our raytracer and assign it to the triangle, similarly to what we did with the opengl rendering. Go ahead and define the constructor for this class, together with initializing this variable.

```

//add to .h
    Material *default_mat;
//add to .cpp
Scene::Scene(std::vector<ThreeDModel> *texobjs,RenderParameters *renderp)
{
    objects = texobjs;
    rp = renderp;
    Cartesian3 ambient = Cartesian3(0.5f,0.5f,0.5f);
    Cartesian3 diffuse = Cartesian3(0.5f,0.5f,0.5f);
    Cartesian3 specular = Cartesian3(0.5f,0.5f,0.5f);
    Cartesian3 emissive = Cartesian3(0,0,0);
    float shininess = 1.0f;
    default_mat = new Material(ambient,diffuse,specular,emissive,shininess);
}

```

Now we can finish "updateScene()"

```

if(obj.material== nullptr){
    t.shared_material = default_mat;
}else{
    t.shared_material = obj.material;
}
triangles.push_back(t);

```

Where we assign the material and push back the triangle into the list. And we are done... right?

Not really. But I am done. From now on, it is your assignment to get an actual raytracer to work. *This is where the fun begins.* Please download the assignment file with the instructions to follow. Good luck!

PS: Please remove the while loop that simulated work in the main raytracer, or things will be really slow!