PH3170: Short Project Report

C++ Ray Tracing

S Bugden

6th December 2014

Abstract

C++ Implemented ray tracer correctly displayed multiple objects within a scene at a specified degree of resolution. Illumination model takes into account ambient, diffusive and specular light as well as shadows on all types of objects. Multiple light sources can be added to give various effects and the colours of all objects can be selected using the standard RGB colour format.

1 Introduction to Ray Tracing

The term "Ray Tracing" is the process of constructing an image using virtual rays of light which pass through a view plane to simulate effects that real light would perform when interacting with other virtual objects. The result is an image which can appear very lifelike, and display a large number of real world effects, such as reflection, dispersion, refraction, shadows and more. A simple example of output from a ray tracer is represented below in Figure 1.

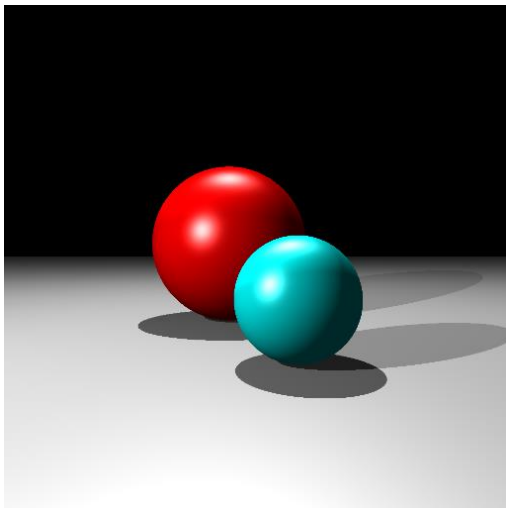

Figure 1: A an image output from this C++ implemented Ray Tracer, showing two spheres resting on a plane, illuminated by two light sources.

The process of ray tracing can be fairly intensive and therefor taxing on a machine when rendering high detailed images as there is a large number of calculations involved with each pixel. This makes ray tracing more suited to tasks where output is allowed ample time to be calculated, such a films and image creation rather than output being needed within very short timescale, such as an interactive game.

Different ray tracers use different illumination models to produce different lighting effects in their images. In this code I have implemented the Phong illumination model[1], which splits light into three components, Ambient, diffusive and Specular.

2 User Guide

This section will illustrate how to use the C++ code to generate images of your own design.
All processes shown here must be placed with the main.cpp file, inside the main function unless otherwise stated.

2.1 Setting up the scene

1. Firstly, set the number of pixels in x and y for your desired image by changing the values for '`pixel_x`' and '`pixel_y`' respectively. These are 1000x1000 by default.

2. Secondly, create and define two `Vector3D` objects which take the coordinate arguments `{X, Y, Z}`, one as the position of your observer and the other as the position of your view plane.

3. Create at least one more `Vector3D` object and give it the coordinates for first point like light source. You can create multiple light sources at this stage if you wish.

4. Create a `Viewplane` object giving the arguments {pixel_x, pixel_y, Width, Height, position}. Here Position has already been defined in step 2 and pixels x and y have been predefined in step 1.

5. Now create the scene which will hold all our objects by creating a `Scene` object and passing in the arguments {Camera position, Viewplane, lightsource1} where Viewplane is the object in step 4 and lightsource1 is your first(and possible only) light source created in step 3.

2.2 Creating objects to place in the Scene

2.2.1 Spheres

1. Create a Vector3D object with the coordinates for the centre point of the sphere you wish to add.

2. Create a Sphere object with the arguments {Radius, Ambient constant, Diffusive constant, Specular constant, Alpha, Red, Green, Blue}. Here the values for ambient, diffusive and specular constant define the properties of the material of the sphere, meaning they will react to light differently, typically they are between 0 and 1, as well as alpha which can be considered "shininess".
   Red, Green and Blue are the colour components of your sphere using the standard RGB model, and must be between 0 and 255.

3. Repeat steps 1 and 2 for multiple spheres if desired.

2.2.2 Planes

1. Similarly to sphere, Vector3D objects are needed, however we need two for a plane, one to hold the position of a point on your plane and another to hold a unit vector which is defined to be the normal of the plane. These two objects define an infinite plane completely.

2. Create a Plane object and pass the arguments { Normal, Point, Ambient constant, Diffusive constant, Specular constant, Alpha, Red, Green, Blue}. Where Normal and Point are defined

in step 1, and the rest is exactly the same to the sphere creation method.

3. Repeat steps 1 and 2 for multiple planes if desired.

2.3 Adding objects to the scene

2.3.1 Adding Spheres

1. Adding a Sphere to the scene is simple at this stage. We have already created our Sphere and our scene, so now all we need to do is use the scenes `AddSphere` function. To do this simply type: `Scene.AddSphere(Sphere);`. Where Scene is the name of your scene and Sphere is the name of one of your spheres.

2. Repeat the code stated in step 1 if you wish to ass multiple spheres.

2.3.2 Adding Planes

1. The Process to add planes to our scene is very similar to adding spheres. The only different here is we use a different, AddPlane, function. To do this simply type: Scene.AddPlane(Plane); where again Scene is the name of your scene, and Plane is the name of one of your planes.

2. Repeat step one for multiple Planes if desired.

2.3.3 Adding Lights

1. Again, very similar to above. Instead here we use an AddLightsource function in replacement for spheres or planes. The code is as follows: Scene.AddLightsource(Lightsource). It is Important to note that this is the function used to add light sources that are not your first. To explain, when we created our scene we used one light source, this function is used if you wish to add any more light sources beyond your mandatory first.

2. Repeat step 1 to add more light sources if desired.

2.4 Starting the Ray Tracer

1. Before your ray tracer can run, it must be told where to place the image when the render is complete. To do this use the scenes SetDirectory function by coding: Scene.SetDirectory("your directory");. Where again scene is the name of your scene object, and directory is written in the standard format.
2. Once you have created and added everything you wish to display in in your image, the last thing to do is start the programme. To Begin the programme simply use the scenes Engage function using the code Scene.Engage();.

2.5 Notes

Depending on the chosen resolution for the image and number of objects in the scene, the ray tracer can take some time to render the image, so a lower resolution may be advised if testing the programme, then Increase the resolution when the image has the desired display. . A smaller loading bar has been implemented to indicate percentage of render complete.

It is important to remember that this ray tracer will attempt to render regardless of whether the objects are placed favourably or not. So think about where you place your objects, make sure they will be visible to your view plane and that they are not too large etc. as they can overlap with each other. Remember planes are infinite! Placing one in front of your shapes will just appear like a wall.

3 Code

3.1 Classes

Vector3D: Basic class which holds three double members, _x, _y and _z which represent coordinates or vector components in three dimensions. The class contains basic vector maths such as addition, subtraction and multiplication. There is also a Dot Product function as well as a magnitude calculation function, which is are used throughout.

Ray: The Ray class is composed of two Vector3D objects. This class is what defines our light rays by containing everything needed for an equation of a line.

Viewplane: The Viewplane class is more complex as it takes 5 arguments to define. Each is a simple concept however, pixels in x direction, pixels in y direction, width and height of your view plane, and then the position, given by a Vector3D object.
The main function of this class is to calculated a Ray object from the observers coordinate to the coordinate of the centre of each pixel on the view plane, this is present in the GetPixelRay functions. Which takes the position of the camera and integers I and j, where I and j define which pixel on the image you want a ray for.

Sphere: The Sphere class creates objects which we want to virtually intersect with in our image, therefore not only are they defined as their shape, (Radius and centre position) but also their material properties such as colour and shininess are defined for use when calculating illumination.
There are three important functions associated with a sphere. The Intersect functions, which check is there is any valid intersection with a ray and a sphere by checking for complex routes. The check shadow function, which check for an intersections with a ray in the positive ray directions specifically. And the Reflect function, which takes a ray, checks if the intersection is valid and then calculates the new ray which reflects off the surface of the sphere.

Plane: Very similar to Sphere class here as both are objects we wish to test for intersections and illuminate if this condition is true. So again, the shape plane is defined (Position on the plane and Normal to the plane) by 2 Vector3D objects, and the properties of the material are added same as the sphere. The Plane class has the same three functions as the sphere class, Intersect, checkShadow and Reflect, however the maths in these functions differs as we are now testing for an intersection with a plane rather than a sphere. Other than that, it performs the same function.

Scene: The scene class holds the main functions of the Ray Tracer and is where rendering takes place. It uses STL containers called vectors to hold all of the objects within the scene and each objects can be called individually. The class contains functions to add objects to these containers in the form of AddSphere, AddPlane etc. In addition to these functions there are two other important functions.
The first is CalcNearestShape, this takes a vector of different shapes and calculated which shape is closest to the viewport, and there is the shape which would be seen, this is calculated for each pixel. The reason this is important is that it stops the tracer from having to calculate ray effects for every shape the ray would intersect, and means it only has to calculate the nearest shape,

saving rendering time, and also displaying the image correctly.
The second and arguably most important functions is the Engage function. This functions runs through every pixel and then through every shape to check whether there is any valid intersections, and if there is compares the reflected ray to each light source and checks weather the location is in shadow or otherwise how it is illuminated by the light to calculate a normalised RGB value of intensity and saves it to a BMP image.

3.2 EasyBMP

The Process in which we output BMP files is through EasyBMP. It is a class which creates a BMP object which has a RGB colour intensity value for each pixel, as well as a specified bit depth of image.

4 Conclusion

The C++ ray tracer performs well when dealing with primitive objects and shows a good representation of real world lighting on these objects. The Code is fairly simple with clean flow from one class to another without cluttering memory, as only the stack is used, and is cleared after each run of a function. The output images in BMP format are well detailed and correctly show each shape.

If I were to improve this project code I would continue to implement new shapes to add to the scene, such as cubes or pyramids. To do this I would implement a Shape template class which would allow each shape class to inherit from this template. I would also add recursive reflection functionality, so each shape could display reflected images of other shapes in the scene on their surface. This would be fairly simple to add and would not take much time, however the Engage function would need to be split up slightly to allow sections to be called individually.

Lastly a highly detailed image, showing multiple plane and sphere objects in a lighted scene to show the possibilities of my C++ ray tracer code, Figure 2.

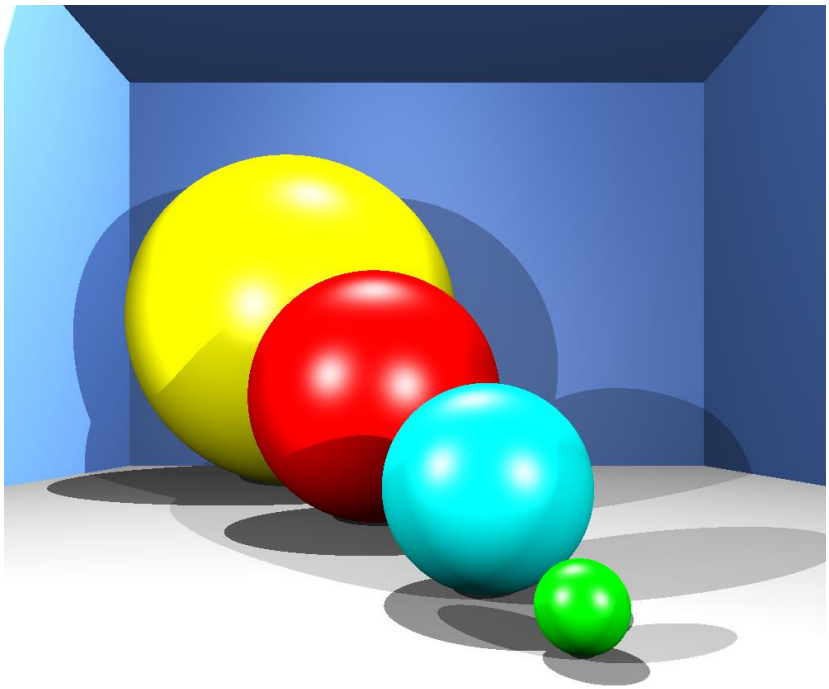

Figure2: A 1000 by 1000 pixel image of 4 spheres surrounded by 5 separate plates, Illuminated in total by 3 separate light sources.

5 References

1. Ray Tracing project script “Ray Script.pdf”, http://moodle.rhul.ac.uk/, December 3, 2014

2. Bui Tuong Phong at the University of Utah, Phong illumination model, http://en.wikipedia.org/wiki/Phong_reflection_model.

3. EasyBMP (Open source) documentation and source code, http://easybmp.sourceforge.net/.