Ray Tracing Program

Hannah Wakeling. Candidate No: 1600184

11th December 2015

Abstract

A ray tracing program was written in the C++ coding language. The functionality of this program implements the reverse ray tracing method to simulate the interactions of light rays with two and three-dimensional objects in a scene. Refraction and reflection methods were written but did not function correctly within the code due to time constraints.

Introduction

In the physical world light rays originate from a source, then reflect, refract and interfere with each other. With each interaction, properties of the light rays change, either in direction, colour or intensity. A program was designed to use optics principles, such as Snell's law of refraction and the laws of reflection [1], to simulate a realistic scene. At the point of submission of this project, the program had the ability to recursively trace rays through a scene, though the functionality of the reflection and refraction was not visible in the final image outputs.

1 Implementing optical physics in C++.

The optical principles of physics can be utilised in code to replicate and simulate real world conditions. To simulate a scenario, one has to deconstruct the elements of the real world, like objects or light rays, into their constituent properties. A real world scenario consists of a space containing light sources that emit light rays which, in turn, interact with the objects stored in that scene. Assumptions can be made to simplify a scenario and therefore code, as has been done in this project. The structure of this project's code allows for easy future development, and therefore better investigations into more complicated situations. Section 3 expands on how these assumptions can be altered to increase the simulation's reality and accuracy. For example, in this project light sources were assumed to be point sources, and the observer was fixed at the origin of the axes. Another key assumption is one of basic shapes. The objects used in the code are three-dimensional shapes that can be described by equations. Fundamentally, light rays and planes consist of two vectors; a light ray can be described using a vector point of origin and a vector direction, and a plane can be described using a vector normal and a vector point on that plane. Spheres consist of a vector position of the centre and a length, the radius. The viewport can be likened to a photograph. The colour of the light incident on each pigment is stored and can be viewed later. It is a rectangle that records an image, so can be described using two lengths and a position. This can be a vector position, but for this stage of the code

it has been assumed that the viewport is at a distance from the observer (the origin) in the y direction only.

Once the scenario has been set up, one wants to investigate how light rays interact with the objects in the scene. To do this effectively a key ray tracing method was used: reverse ray tracing. This is the simulation method of working backwards from each pixel following where a light ray would have come from, as shown in Figure 1. This is more practical than the opposite: trying to simulate the paths and interferences of every

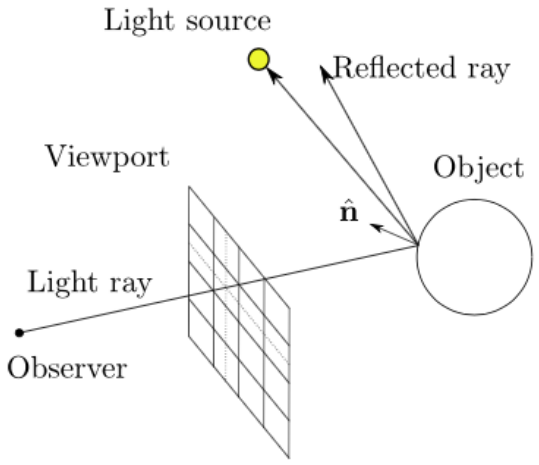

Figure 1: Diagram of reverse ray tracing [2].

possible light ray in a scene. The majority of light rays would not hit one's viewport therefore.

2 The C++ Program and the Methodology

2.1 The Classes

The `Scene` class consists of vector containers that hold pointers to the objects in the scenario. There are two vectors defined in this class, one for three dimentional shapes in the scenerio, and one for the light sources. The vectors can be altered as the user wishes using methods within the class. As the program automatically destroys the information stored at the end of the run, it was elected to not destroy the objects from the heap. The information should not be destroyed until after the program has created the image, which is the final output of the code.

The `LightRay` class describes a light ray. It consists of two properties, it's vector point of origin and it's vector direction. The method `Vector3D calculatePointOnRay(float lambda)` returns the point on the light ray that intersects with the object using lambda, the solution of the intersect calculated by the `checkIntersection(LightRay &light_ray)` method for each shape.
`float calculateMagnitude(float lambda)` returns the magnitude of the light ray from point to intersection.

The `LightSource` class includes position and intensity. Intensity is limited to be between 0 and 1, with exceptions throwing the integer value of 2. There is the possibility for a light source to have colour, but this not currently fully implemented within the project. A light source can be given area through creating an

array of light sources.

The `Shape3D` base class utilizes polymorphism with two methods. The calculations that these methods perform are dependent on the shape and therefore need their own definition within the shape's class. The other methods calculate or return the desired properties for all shapes.

The `Sphere` and the `Plane` classes inherit the methods from Shape3D. If the reflectivity value given is not between 0 and 1, an integer value of 1 is thrown for exception handling. For future expansion of the code, an overloaded function has been created to include the refractive index of the objects. As mentioned above, these classes define the virtual functions declared in Shape3D. Other shapes can simply be added to the program by inheriting Shape3D.

The `Vector3D` base class contains mathematical methods of manipulating vectors. A vector is defined as an array of three elements. A bug was located with the use of multiplyBy within the overloaded operator for multiplication. It changes the original vector, which is not the desired effect. This was corrected for the whole program, apart for its implementation in the Sphere class method `checkIntersection`. For the spheres to be visible in the final output the bug has to be kept in that instance. It is thought that this is one reason behind the reflection method not having the desired effect on the final output (one sphere is not visible in the surface of other spheres). To debug this, it was investigated at which point the direction of the light ray is used in the sphere calculations. In the future development of this code, to potentially rectify the problem, one would have to identify why the light ray intersects with the sphere, but the output does not display an image of the sphere.

The `Colour` class. Inherits from Vector3D and has the vector element values limited to valid RGB values. If the values are outside of the range, an integer value of 0 is thrown for exception handling.

The `Viewport` class is the part of the code that implements most of the other code written. It contains two methods, `render(Scene &scene)`, and `traceRay`. It will retain the correct ratios to avoid undesired effects like fish-eye, and is setup as seen in Figure 2. The `render(Scene &scene)` method initiates a light

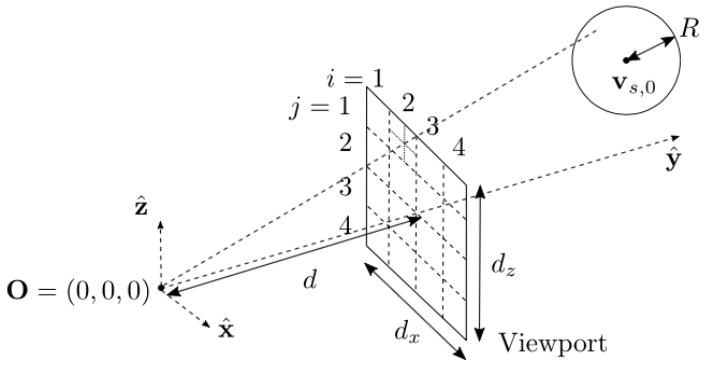

Figure 2: Diagram of the application of the viewport [2].

ray from the origin in a direction that passes through a pixel. It does this for each pixel using a nested for loop. It has the additional property of antialiasing, which diminishes the effects of converting analogue data to digital. This however increases the number of rays produced per pixel and increases the run time dramatically. It calculates the RGB colour of the pixel using `traceRay` and assigns this value to the pixel using the EasyBMP Cross Platform Windows Bitmap Library. `traceRay` is a recursive function. It iterates through each object stored in the vector container, checking whether the ray produced intersects with them.

For each intersection the magnitude between the light source and the intersection point is calculated. The vector iterator is returned to the shape pointer with the smallest magnitude. It then iterates through each light source, calculating the intensity of the light at the point of intersection, and adds to a final intensity value. It averages the contributions of the light sources, then calculates the current colour of the pixel. It then recurses using the reflected (or refracted) light rays, up to a limit set by the user. It returns the total colour (that will later be assigned to the pixel).The reason for the limit of the recursion is to keep the program run time down. When developing the code, an alternate option to setting the recursion limit manually is to give the light ray a maximum trace depth, or to stop when light ray hits a diffuse object.

There is a bug that reduces the depth effect of the image output. The spheres appear to intersect when occupying the same pixel, even though they do not intersect in the vector space. Investigating this effect seems to indicate that there is either an error in the calculation of the effect of the reflected rays, perhaps located in the recursion of the `traceRay` function or there is an error in the use of the vector space, which means that the spheres are in fact intersecting.

The `main` has the exception handling and some useful colours defined, which could be stored in STL containers; for example the colours would be a map of vectors.

2.2 Guide to using the code.

The code has been made as user friendly as possible. To create an image, one would only have to follow the few simple steps below. These come with a snippet of sample code for extra clarity. This would have to be placed within `try{}` to allow for exception handling.

1. Create a scene in the heap memory and a pointer to it.
   ```
   Scene* s = Scene();
   ```
2. Create object(s) in the heap memory and pointers to them; spheres, and planes have been defined, but there is the possibility of adding other shapes.
   ```
   Sphere *sphere = new Sphere(float rad, Vector3D centre, Colour colour, float reflectivity);
   ```
3. Add these object(s) to the scene.
   ```
   s->addSphere(sphere);
   ```
4. Create light(s) in the heap memory, with a pointer to it.
   ```
   LightSource* l = new LightSource(Vector3D position, float intensity);
   ```
5. Add the light soure(s) to the scene.
   ```
   s->addLightSource(l);
   ```
6. Create the viewport in the heap memory and a pointer to it. Set the desired dimensions and distance of the viewport from the observer. `x` and `y` are the dimensions of the viewport width and height, respectively, and where `distance` is the distance of the viewport from the origin point in the `y` direction, the point at which the observer would see the objects from.
   ```
   Viewport* v = new Viewport(int nx, int ny, int d, Scene &scene);
   ```
7. Render the image of the scene.
   ```
   v->renderImage(*s);
   ```

3 Future development possibilities of the code.

The code has been structured to allow for easy development and debugging. This includes it containing multiple methods that are not yet implemented in the program, for example 'get' functions exist for most of the protected or private data of the classes.

There are many ways in which this project can be expanded and refined, including; adding simple things like other shapes, adding functionality to import files of predetermined scenarios, implementing a gas of particles for diffuse scattering, using complex lights that have a fall off value that determines the narrowness of beams or adding the physical effects of diffraction, polarisation, Monte Carlo effect from the source to the user, Phong lighting.

One addition that was focussed on was the production of shadows. This would have been added to the `traceRay` method. The potential theory behind this is to create a vector in the reverse direction to an incident light ray on an object. If this intersects with anything other than a light source, the light intensity would be diminished, and so creating the effect of shadowing.

When choosing the sizes of the viewport and aspect ratios they had to be just right to give an image as one would see in the real world. These can be manipulated to give effects like fish lensing.

4 Summary

The ability to infinitely expand and enhance code was the key aspect to taken from this project. Throughout the three weeks allocated it was made clear that by creating a project with a concise structure and well defined functions and classes, the code can be continuously altered and improved. A wide variation of coding techniques were used to produce a ray tracing program. The use of program control structures, the Standard Template Library, pointers and references for memory representations of variables were essential in the project. In addition, the use of classes with polymorphism and inheritance structured the code in such a way that it was much easier to navigate and freely develop the code by readily adding methds etc. Finally, Figure 3 and 4 display some of the possible final outputs of the program, indicating successful manipulation and implemention of ray tracing in C++.

References

[1] Nave R. *Refraction of Light, Snell's Law.* [Online] Available from: `http://hyperphysics.phy-astr.gsu.edu/hbase/geoopt/refr.html`

[2] Boogert S. *PH3170 (C++ Programming) Project 1 : Ray tracing program.* [Online] Available from: `https://moodle.royalholloway.ac.uk/mod/resource/view.php?id=173975` [Accessed 1st December 2015].

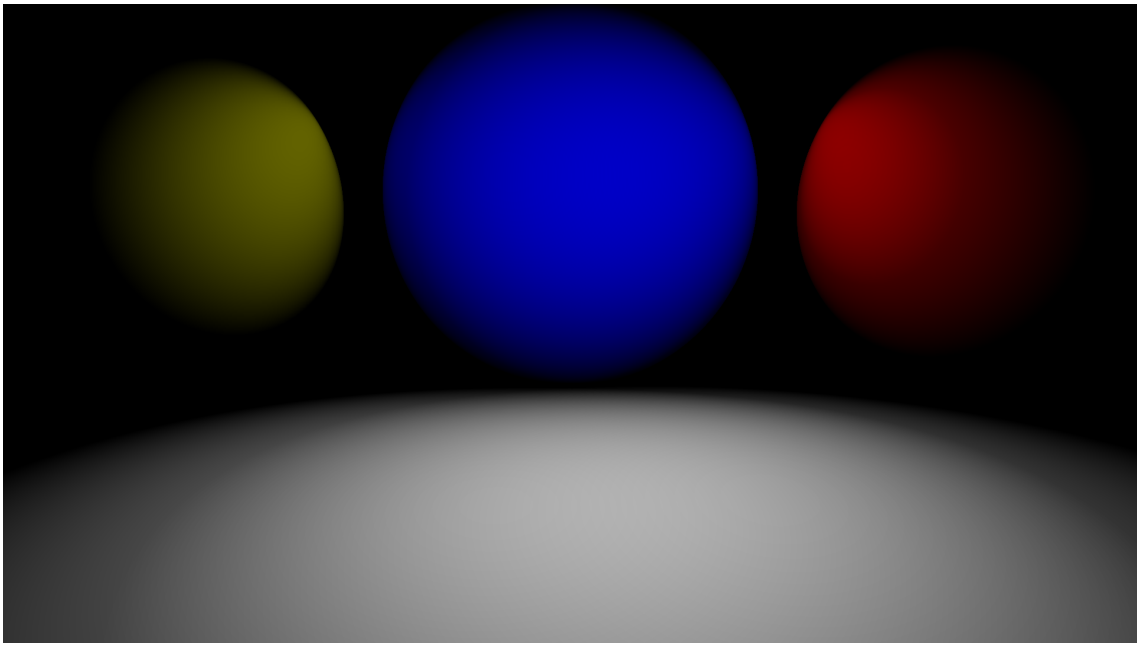

Figure 3: Possible image created from the ray tracing program submitted with a plane and 3 spheres.

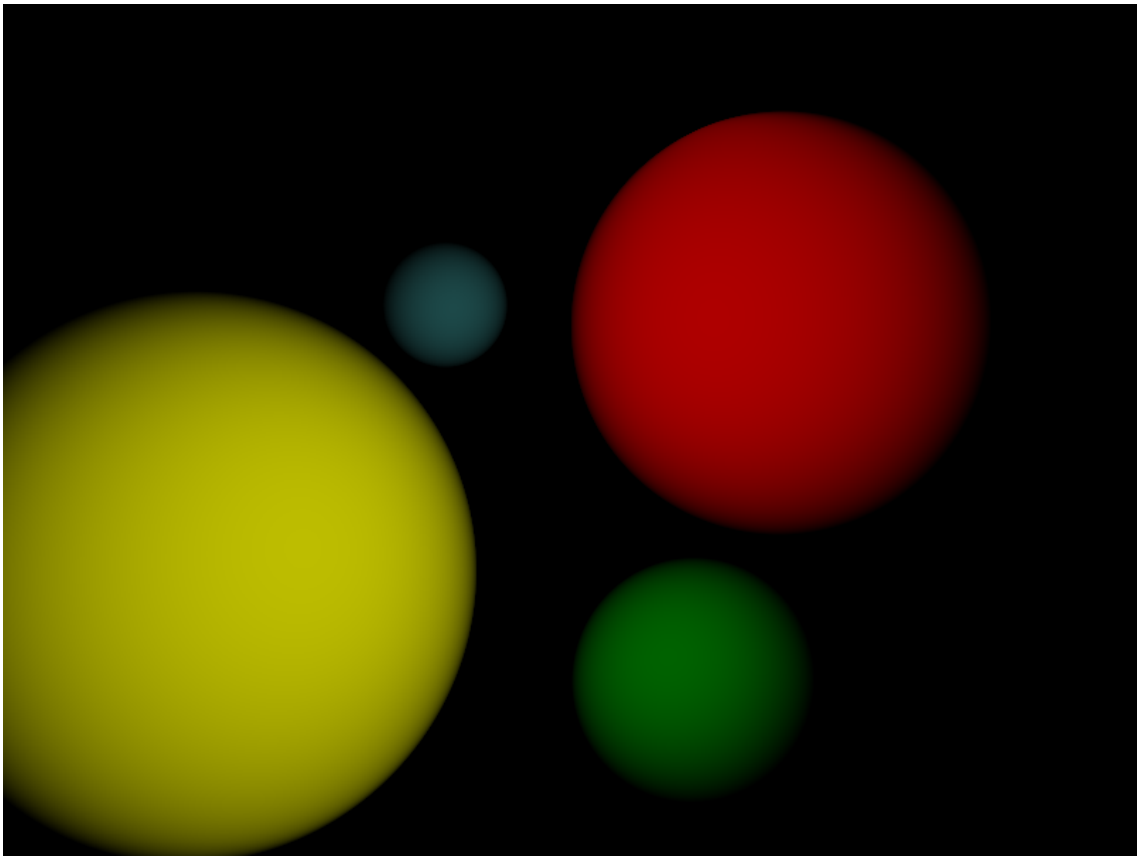

Figure 4: Possible images created from the ray tracing program submitted with 4 spheres.