

# CA1 ASSIGNMENT

## FINAL REPORT

CVA LEVEL 5 COMPUTING FOR ANIMATION

Quentin Corker-Marin

i7624405

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Program Structure</b>	<b>1</b>
<b>3</b>	<b>Algorithms</b>	<b>1</b>
3.1	Overview . . . . .	1
3.2	Ray Tracing . . . . .	2
3.3	Ray Generation . . . . .	3
3.4	Ray/Triangle Intersections . . . . .	4
3.5	Shading . . . . .	5
<b>4</b>	<b>Program Output</b>	<b>5</b>
<b>5</b>	<b>Possible Expansions</b>	<b>12</b>

# 1 Introduction

When I was first designing my ray tracer, I was planing on implementing a physically based lighting and shading system. Unfortunately I didn't give myself enough time to fully understand the maths and physics behind it so I was unable to implement the physically based parts. Instead my ray tracer uses a mixture of the phong reflection model and ray traced reflections and refractions. It was a shame that I didn't manage to implement PBR and in retrospect I would have found it more interesting to do a very simple sphere ray tracer and have the focus on the lighting and materials.

## 2 Program Structure

When I was designing and during the early implementation, I was using the book *Physically Based Rendering: from theory to implementation* to guide the programs structure. This meant that the design was quite focused on producing an expandable system. All intersect-able objects in the renderer (`Scene`, `Primitive`, `GeometricPrim`, and the `TriangleMesh`) inherit from `Primitive` so that they all have the same intersection interface. While I didn't have time to add all of the expansions I wanted, this structure means that it would be easy to do so. For instance, to add a new type of intersect-able geometry, I would have to write its intersection methods and add it to the parser and it could be placed inside a `GeometricPrim` without having to alter the class or the renderers intersection methods.

I also split the `Triangle` class, creating a second `TriangleData` class that holds all of the data that is required for shading the triangle, but is not needed for intersection checks. This was an attempt to make the program more cache efficient by only having the necessary data in the class that is checked for intersections. This means that less unnecessary data is gathered by the prefetcher and only accessed when an intersection is confirmed and the shading data is required.

## 3 Algorithms

### 3.1 Overview

The program is run through the `Viewport` class which connects the ui to the main renderer. The ui is linked to a `RenderSettings` structure which passes

the current values in the ui to other classes when the buttons are pushed. When the program is being run, the first thing that should happen is the user loads a scene to the program through the *Load Scene* button. When this happens the scene path is passed to the parser which, if the scene file is found, translates it and constructs a **Scene**. Once this happens, the **Scene** can be passed to the renderer using the *Render* button. When the **Renderer** is run, it splits the final image up into tasks. Each task is a group of pixels (width and height of groups are set in the ui) to be rendered. By splitting up the rendering into separate tasks, it allows them to be run in parallel using the `std::thread` template. A ray is constructed for each pixel in the final image and are traced through the scene to find the colour for that pixel. The final colours are stored in the **Film** class to be rendered to the screen using **SDLWindow** and saved. When the window is closed, a new scene can be selected or ui setting changed and the same scene re-rendered.

### 3.2 Ray Tracing

Ray tracing is a well known algorithm and, in my program, happens inside **RenderTask** objects. The source code for my ray tracing can be found in **TenderTask.cpp** and the pseudo code for how it works is below. It is split up into 2 functions: render, trace.

```
render()
    for each pixel in output image:
        construct a ray from the camera to that pixel
        out colour = trace( ray )

trace( ray )
    init outCol to (0, 0, 0) //r, g, b
    check the ray for intersections with the scene geometry
    if an intersection is found:
        find out if the ray is going from inside to outside or outside to inside
        find ratio of refracted and reflected light contributions
        if current ray bounce < max ray bounces:
            if material is reflective:
                generate reflected ray
                reflected col = trace( reflected ray )
            else:
                reflected amount = 0
```

```

    if material is transparent:
        generate refracted ray
        refracted col = trace( refracted ray )
    else:
        refracted col = blinn shading( intersection )
else:
    refracted col = blinn shading( intersection )
    reflected amount = 0
outCol = refracted col * (1 - reflected amount)
        + reflected col * reflected amount
return outCol

```

### 3.3 Ray Generation

Ray generation happens inside the **Camera** class when it is passed an x and y value that represent a position in the final image in pixels. The method takes x and y as doubles because it needs to be able to generate rays for values in between the discrete pixels to be used for anti aliasing. rays are generated in camera space, and then converted to world space when it is passed out of the function. The process for generating rays from the camera is as follows:

```

generate ray( x, y ):
    init origin to (0, 0, 0)
    convert x and y to normalised display coordinates (0.0 - 1.0)
        to do this x is divided by the screen width
        and y is divided by the screen height
    x and y are then converted from NDC to screen space
        to do this x and y are converted from the range 0-1 to -1 - 1.
    x and y are then multiplied by the fov multiplier
    x is also multiplied by the aspect ratio
    a vec4 is then constructed with (x, y, 1, 1)
    this is the direction vector of the ray in camera space
    this is multiplied by the camera to world rotation matrix
    the origin is multiplied by the camera to world transformation matrix
    these are then used as the origin and direction of the new ray to be
        passed out

```

### 3.4 Ray/Triangle Intersections

The ray triangle intersections are the most performance critical part of the program. Every ray that is generated must be checked for intersections. I therefore looked around for a while before I found one that seemed to be popular. I found the algorithm at <http://graphics.stanford.edu/courses/cs348b-98/gg/intersect.html> and it works as follows:

```
intersect( ray, triangle ):  
    calculate the point of intersection between the ray and the plane  
        this is done by equating the parametric form of the ray ( $O + Dt$ )  
        to the equation of the plane made up by the triangles ( $N \odot P + d = 0$ )  
        this is then rearranged to find the parameter  $t$  which represents the  
        distance along the ray that the intersection happened.  
    the intersection point is then checked to see if is inside the triangle  
        the triangle is projected along its major axis  
        this is the largest value of its normal  
        this reduced the problem to 2 dimensions  
     $V_0$ ,  $V_1$  and  $V_2$  are the triangle vertices  
     $P$  is the intersection point  
    3 vectors are then constructed:  
         $vec\_a = V_0 \rightarrow P$   
         $vec\_b = V_0 \rightarrow V_1$   
         $vec\_c = V_0 \rightarrow V_2$   
    these are used to get two values  $\alpha$  and  $\beta$  which represent  
     $vec\_a$  in terms of  $vec\_b$  and  $vec\_c$   
     $vec\_a = \alpha vec\_b + \beta vec\_c$   
    if  $\alpha > 0$  and  $\beta > 0$  and  $\alpha + \beta < 1$ :  
        the  $P$  is inside the triangle  
    else:  
         $P$  is outside  
     $\alpha$  and  $\beta$  can also be used to interpolate other values  
    from  $v_0$ ,  $v_1$  and  $v_2$  such as uvs and normals
```

The actual implementation of this algorithm that I used is explained in the link above and has increased efficiency.

### 3.5 Shading

To generate the colours of materials I used the Blinn shading model. This involves adding together ambient, diffuse and specular contributions from each light. This is done as follows:

```
blinnShading( intersection ):
    out colour initialised to (0, 0, 0)
    out colour += ambient light * material colour
    for light in scene:
        if light is visible from intersection point:
            outCol += specularShading()
            outCol += ambientShading()
    return outCol

ambientShading():
    //lambertian reflection
    return normal  $\odot$  light dir * light intensity * attenuation * mat colour * light colour

specularShading():
    //blinn reflectiion
    half vec = normalise(1 + v)
    spec = max(0, normal  $\odot$  half vec)
    return spec ^ material shininess * light intensity * attenuation * light colour
```

## 4 Program Output

This section contains a history of what the program was outputting as I was writing it.

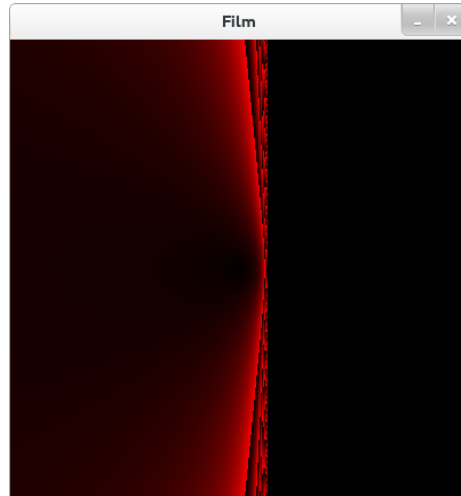


Figure 1: First image produced by the renderer using a naive ray generation method that was wrong

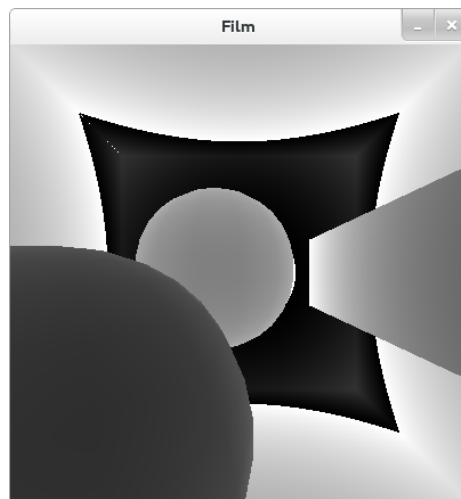


Figure 2: Once the rays were being created properly, I rendered a depth image, but I did not clamp the depth value so when it got above a certain range it was lost

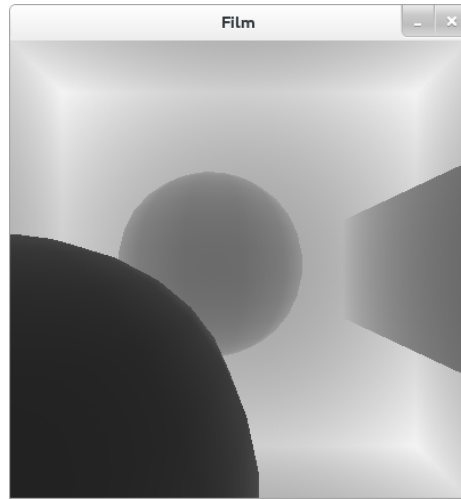


Figure 3: Depth properly clamped

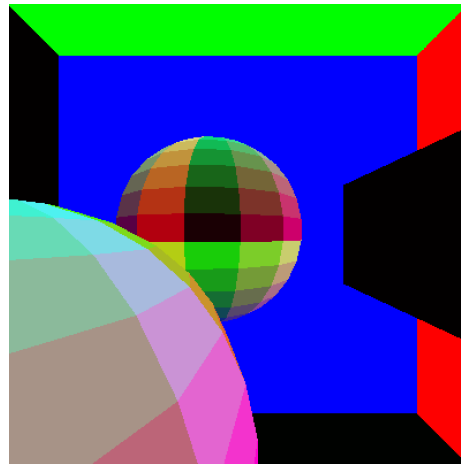


Figure 4: I then tried shading the geometry using the normal vector at that point, with odd results



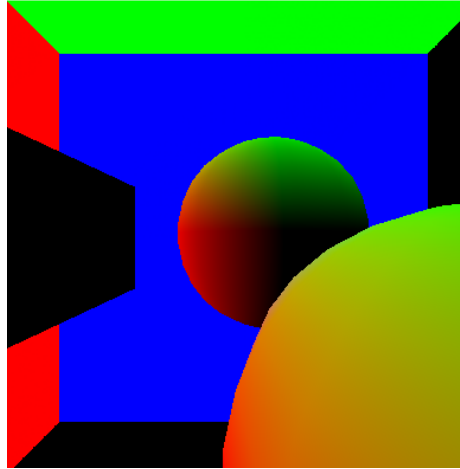


Figure 5: I then changed to interpolating the normals, producing a much nicer result. This also mean that I could interpolate any values associated with the triangle vertices across the face

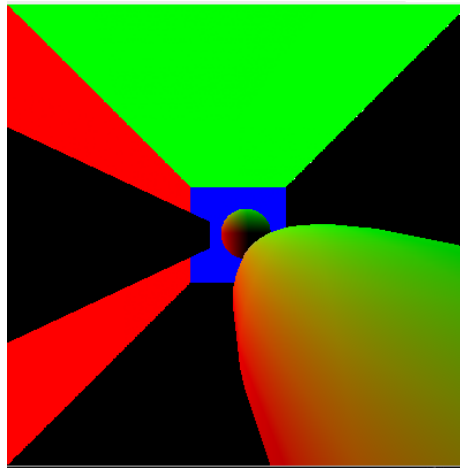


Figure 6: Implemented fov

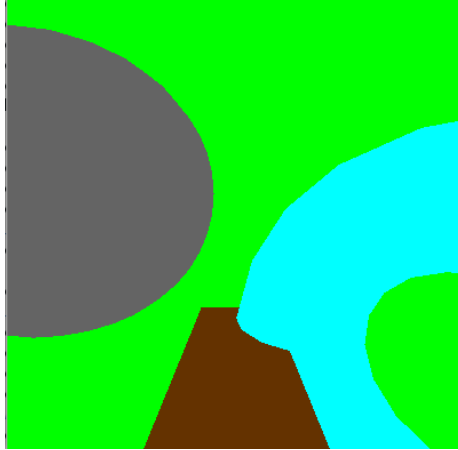


Figure 7: I then gave each object in the scene a flat colour so that I could start to test the material class. At this point I was using hard coded meshes, I wrote a python script to translate an obj into the hardcoded scene which I pasted into the program

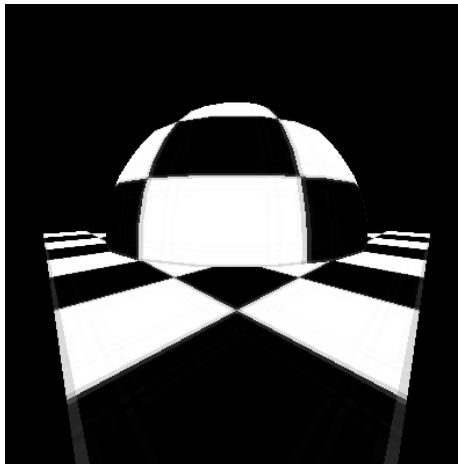


Figure 8: Once I had the flat colours working, I implemented the texture class so that the colour was found by interpolating uvs and accessing those texture coordinates

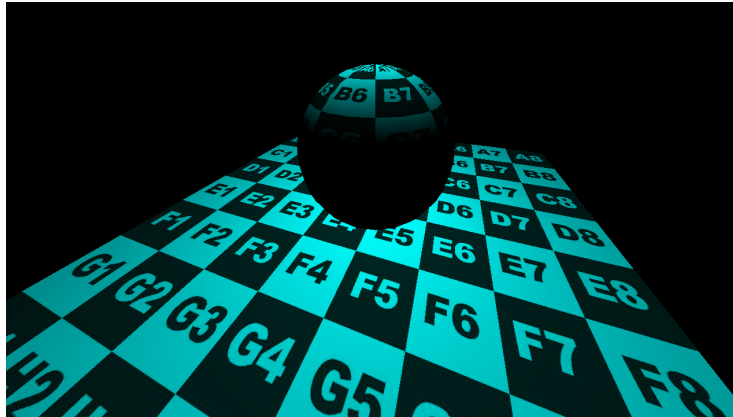


Figure 9: I then started working on the shading, implementing the lambert shader

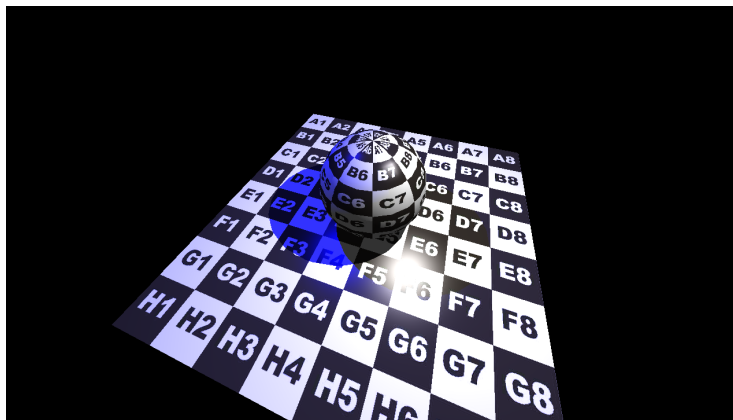


Figure 10: I then added shadows and specular highlights to the shading

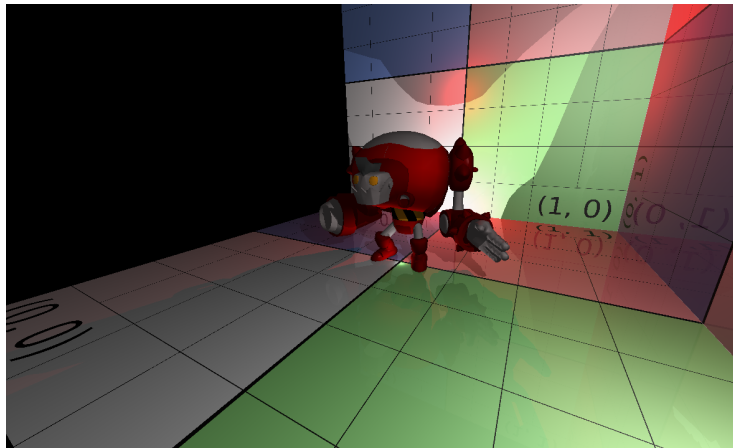


Figure 11: I then added ray traced reflections and refractions. By now I had implemented the parser, which meant I could read obj files straight into the program, without hard coding them

## 5 Possible Expansions

Because of the way that the program is written, implementing new types of geometry is as simple as writing a new class that inherits from primitive, implementing intersection routines for it and adding it to the parser. In this way, functionally represented objects could be included such as nurbs curves, spheres and planes. I would also like to redo the rendertasks rendering loop to separate shading into its own class. This would allow for different types of shading to be implemented for different materials, including possibly proper physically based shading.