CA1 Assignment

Final Report

CVA level 5 Computing for Animation

Quentin Corker-Marin
i7624405

Contents

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 my self 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 raytraced 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 starting out, I was using the book *Physically Based Rendering: from theory to implementation* as my guide to the structure of a renderer and that lead to a lot of my design decisions being focused on making an expandable system. This meant that I probably wrote more classes than I needed to for what I was doing. For instance I have the abstract base class `Primative` which is inherited only by the `GeometricPrim` class, which holds triangular meshes in my final program. The idea was to implement an `AggregatePrim` class that would hold primitives in an acceleration structure for faster ray intersections and any other mesh types would have also used their own `Primative` container. Unfortunately I wasn't able to implement these and so some of my classes seem to be wasted. This did however, have the advantage of teaching me what sort of things I would need to think about when designing a larger program and leave my ray tracer in a state where some extensions could easily be added.

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 3 functions: render, trace and calcShading

```
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 cordinates (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
  this is multiplied by the camera to world rotation
  the origin is multiplied by the camera to world transformation
  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⊙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
    V0, V1 and V2 are the triangle vertices
    P is the intersection point
    3 vectors are then constructed:
      vec_a = V0->P
      vec_b = V0->V1
      vec_c = V0->V2
    these are used to get two values α and β which represent
    vec_a in terms of vec_b and vec_c
    vec_a = α vec_b + β vec_c
    if α > 0 and β > 0 and α + β < 1:
      the P is inside the triangle
    else:
      P is outside
    α and β can also be used to interpolate other values
    from v0, v1 and v2 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 ⊙ light dir * light intensity * attenuation * mat colour * light

specularShading():
    //blinn reflectiion
    half vec  = normalise(l + v)
    spec = max(0, normal ⊙ half vec)
    return spec ^ material shininess * light intensity * attenuation * light colou
```

4 Program Output

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

In this first image, I had a naieve and wrong ray generation method. The read was meant to be set by the intersection distance from the camera

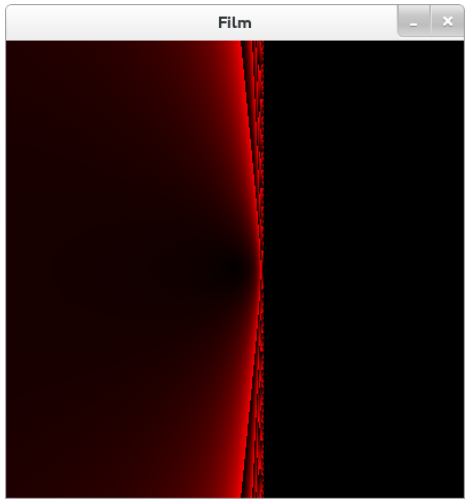

Figure 1: First Image produced by the renderer

Here I hadn't clamped the maximum distance to white so it looped around

to black again

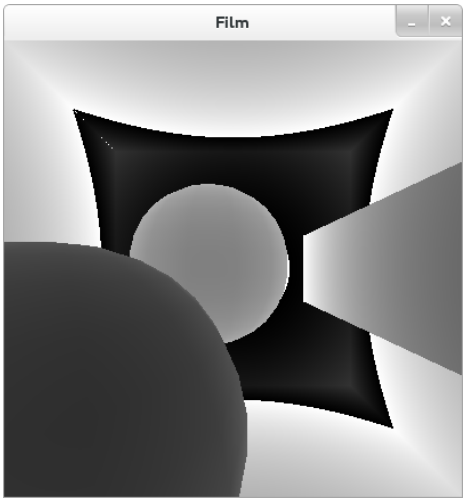

Figure 2: Broken depth

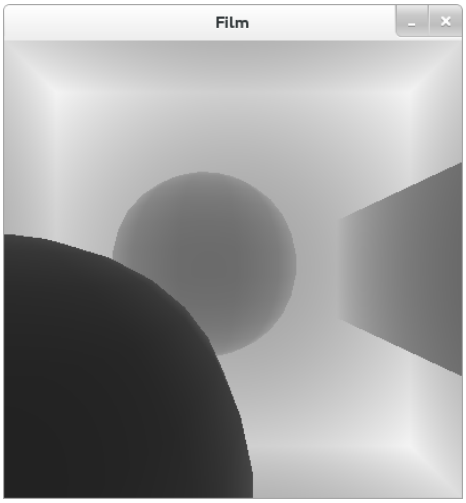

Figure 3: Depth properly clamped

I then went on to set the colour using the normal of the intersection point

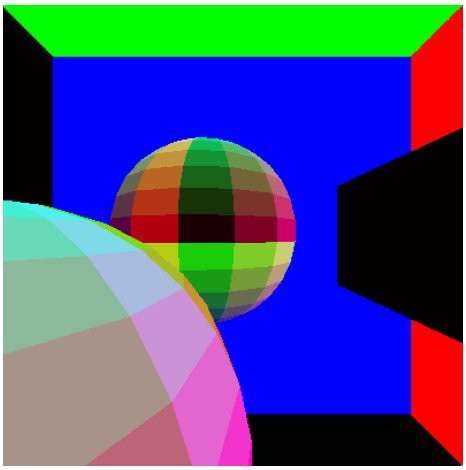

Figure 4: Normals!

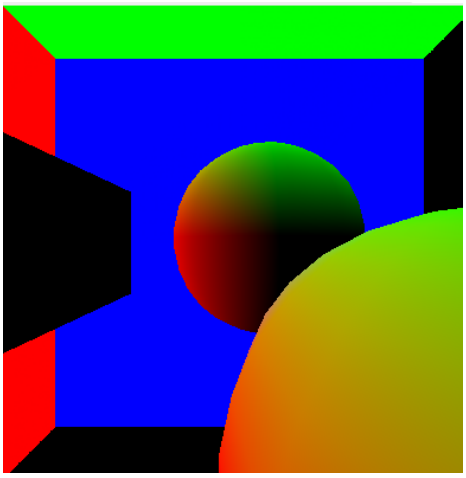

Figure 5: Interpolated normals

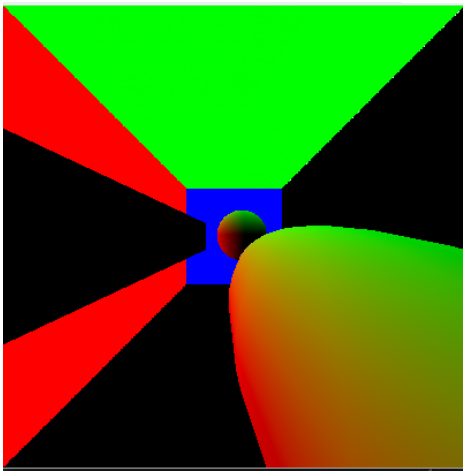

Figure 6: Implemented fov

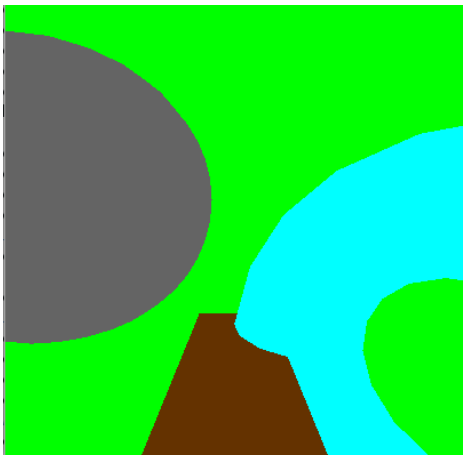

Figure 7: Materials have colour

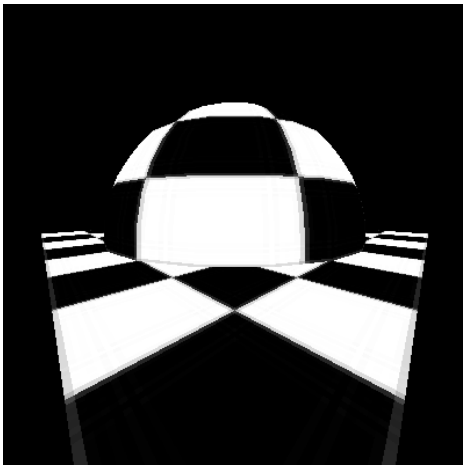

Figure 8: Reading diffuse colour from texture files

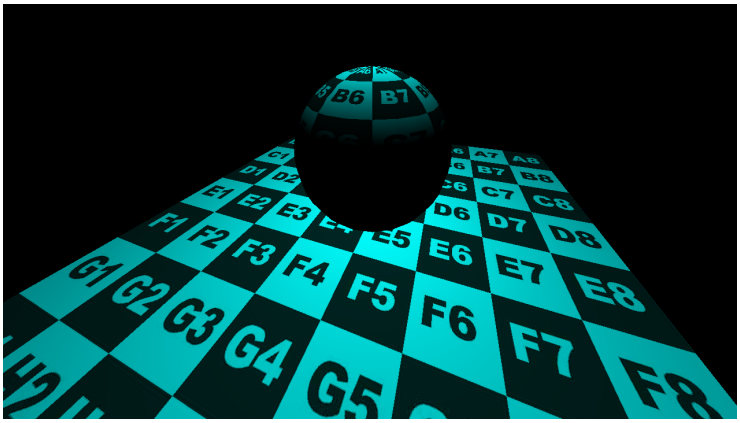

Figure 9: Lambertian reflection

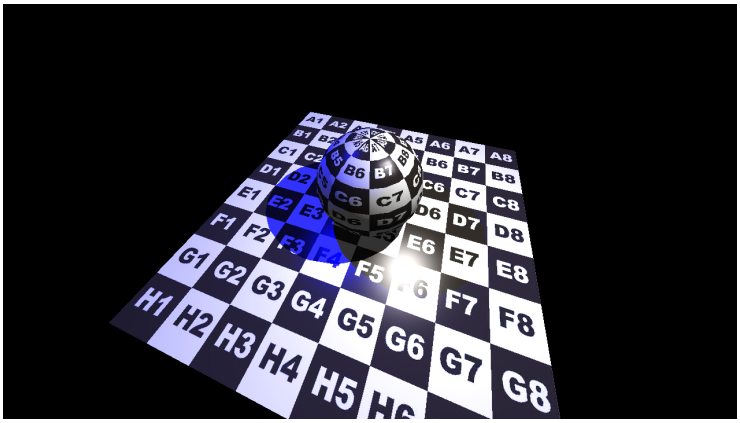

Figure 10: Shadows and Specular

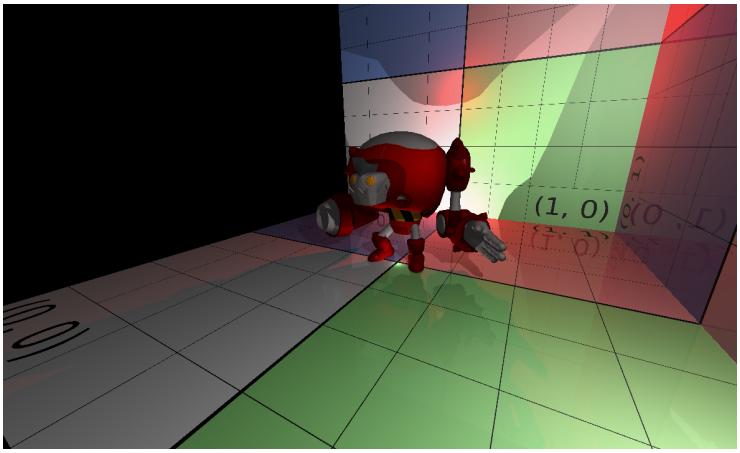

Figure 11: reflections and reading objs

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.

6 Conclusion