Project Report:

Creating a Real-Time Raycasting Rendering Engine

Author: Simon Titcomb
Supervisor: Dr. Frank C Langbein
Moderator: Dr. Jianhua Shao

ABSTRACT

In this report I investigate whether it is possible to create a real-time raycasting rendering engine by utilising the GPU to perform the ray intersection algorithm and a lot of the other computationally expensive calculations.

I will first outline my goals for the project and give the general overview for the project in the introduction. Then I will give some background information on the main problem of the raycasting technique and why it’s not used for real-time applications. I will also talk about some other rendering engines that use raycasting and give a brief overview of the third party libraries I use in my application, along with my justification for doing so.

After the background information I will go into depth about how I actually went about creating the raycasting rendering engine and how I made attempts to keep it performing in real-time. I will go through every major iteration of program and talk about the performance impact after each feature was added, along with the pros and cons of adding the feature.

Once I have covered the implementation I will evaluate the final version of the application and discuss the various capabilities of the program, as well as it’s shortcomings. I will then discuss the future improvements I could make and reflect on all that I’ve learned throughout the project, I will then finish with my conclusion of the project.

INTRODUCTION

This project is an investigation into whether it is feasible to use a raycasting rendering approach rather than the traditional rasterisation method for a real-time application. Traditionally raycasting is much slower than rasterisation, but with new techniques and hardware available, things like programmable GPUs mean that the raycasting calculations could be moved onto the GPU to achieve a real-time rendering method. I investigated whether it was possible to program the GPU to perform these calculations and to what extent I could utilise the GPU to perform even more calculations, thus removing the burden fom the CPU.

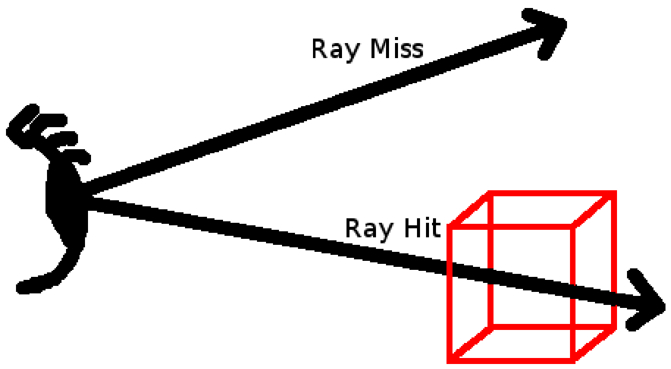
For the course of the project I had set several goals that would register as milestones for how far the application had come. My first goal was to create a raycasting engine that could render a simple scene. A simple scene in this case being a single simple geometric shape, such as a triangle or a square.
Once this goal was reached my next goal was to see how the application coped with rendering a multitude of these objects. I would probably have to implement some kind of optimisation technique in order to cope with a number of the objects so I would have to research into ways of achieving this.
Then when the application was able to render lots of simple geometry I planned to add a little more detail into the scene, making it slightly more complex and a lot more interesting. I planned to use shading to do this, and maybe with enough time I would come back and add more complicated details such as reflections. The next step after this however would be trying to render a more complex object. I had two options, I could either try to render more complicated geometric shapes such as spheres and cylinders or I could try and implement ray triangle intersections in order to load external model geometry and render these.
Once the application was capable of rendering complex models, at decent speeds so optimisation along the way would probably be necessary, I planned to again improve the look of the application, probably by using textures associated with the external model. I would consider the application complete after this as the main goal of being able to render a complex object at a decent framerate will have been achieved.
If time permitted I planned to further optimise and add more detail to the scene by adding in more interesting lighting or reflections.

All of the data obtained regarding framerates and performance was run on the same machine for consistency. Although the application is capable of running multiple hardware I chose a relatively high performance machine to test the application on. This was to ensure that the hardware had the latest graphics drivers as the techniques used in the application require the newest OpenGL specification for which support is potentially limited on older hardware. The specifics of the hardware are below:

OS: Windows 10
Memory: 16GB
CPU: i7-4790K @ 4GHz
GPU: NVIDEA GeForce GTX 980 (4GB RAM)

BACKGROUND INFORMATION

Raycasting is a simpler and much quicker version of raytracing, which is a rendering technique used to render computer graphics onto a screen. It uses ray-surface intersection tests to determine whether a ray sent from the ‘camera’ into the ‘world’ hits an object.



Raycasting is more simplistic and quicker than raytracing as it does not recursively trace additional rays that sample the radiance incident on the point that the ray hit. This means that once a ray hits an object it returns, it doesn’t then trace off from that point to see if it can hit another object and thereby adding additional colour via methods such as reflection. Having said this raycasting is still much more computationally expensive than the traditional approach of rasterisation due to the nature of the technique. A ray is sent per pixel for the screen resolution, so a high resolution means even more computations. Even at small resolutions there are potentially hundreds of thousands of rays being sent, each having to test if it has intersected with an object in the scene, and of course the more objects there are the greater the computation expense becomes. This expensive computational cost is the reason raycasting is not widely used within real-time graphical applications, typically techniques such as this are used offline and then the results rendered at a later stage.

Hardware has now progressed to such a state that we have dedicated GPUs capable of processing not only graphics directly, but that are programmable to compute things for us instead of relying solely on the CPU. As such we are now capable of utilising such computationally heavy rendering techniques in real-time as we can push a lot of the calculations onto the GPU, freeing up the CPU for other tasks. It is now feasible that real-time raycasting could be achieved at certain resolutions for non-complex scenes, and potentially even greater resolutions with complex scenes.

There are a few examples of teams trying to achieve real-time raycasting or raytracing techniques. One of the most notable is the Brigade engine which is *“a real-time rendering engine for video games. It uses path tracing to render images as opposed to rasterisation like most other 3D game rendering engines. Path tracing is an extension of the ray tracing algorithm. It simulates many light paths per pixel and takes the average value to calculate the final colour of each pixel. Whenever a ray hits a surface, a new ray is traced from that* *hitpoint in a random direction until the maximum path depth is reached or until a Russian roulette-like mechanism kills the ray. As a result, Brigade is able to produce effects like diffuse colour bleeding, glossy (blurry) reflections, soft shadows, real area lights, true depth of field, and much more."* Whilst the engine makes use of a slightly different technique in path tracing, it is not too dissimilar to my project. Clearly for my project there is not sufficient time to generate high quality lighting effects or to render such massively complex scenes; the aim of the project is to explore the basics. However the Brigade engine is a good example of what can be achieved and what I would aim for had I the time.

There are a couple of approaches to creating a raycasting rendering engine. The original method was to perform all of the ray intersection calculations on the CPU and record the colour value for each pixel so that they could be sent to the GPU. This approach does not lend itself well to rendering in real-time as it is so computationally expensive that it's not possible to perform the calculations just using the CPU quickly enough to render at a suitable framerate. Instead another approach is to use the GPU to perform all the ray intersection calculations and free up the CPU for other means. GPUs have become fast enough that it's possible to use them for these calculations along with the rendering and still perform in real-time. This is the approach that the aforementioned Brigade engine uses, and the approach I will be taking in this project.

In order to achieve the real-time rendering I utilised several third party libraries, these were mostly used to set up and make working with OpenGL a lot easier.

GLFW: I used the GLFW library for window management as it is the window management library I am most familiar with. It also has the benefit of being cross-platform so my application will be compilable with platforms other than Windows.

SOIL: The SOIL library was used for loading images into the application as it has minimal overhead and like GLFW it is platform independent, ensuring that the application can be compiled on other platforms.

ASSIMP: In order to load model files and their data into the application so that more complex models can be rendered rather than just simple geometric shapes I used the third party library ASSIMP. Again like all of the libraries I'm using, ASSIMP is cross-platform allowing for greater access to my application.

GLM: GLM is an extremely useful header only library that I used to simplify matrix and vector calculations. It has the advantages of being easy to use and very lightweight, and since I have used this library extensively I am very familiar with it.

GLEW: GLEW was used to load all of the OpenGL functions required, although this had no impact on the application itself apart from making working with OpenGL a lot easier. GLEW is core for developers wanting to use OpenGL and as such I am very familiar with the library.

IMPLEMENTATION

When I first started the project I had a rough idea of how raycasting worked, but had to research how to move the calculations onto the GPU instead of the CPU. I discovered two possible routes to achieve this. I could use OpenGL and utilise the API's new "compute shaders" for the calculations, or I could use a GPGPU approach using NVIDIA's API CUDA or OpenCL to perform general purpose computing on the GPU directly.

I ended up deciding to use OpenGL and the new compute shaders as I already had significant experience with OpenGL so development would be significantly easier having already used the API. Although going a GPGPU approach may have yielded better results, I am unsure the magnitude of the difference would be huge. As an aside OpenCL and definitely CUDA are not as widely supported as OpenGL, so the application would potentially be unable to run on specific hardware.

All of the ray intersection code is written in the GLSL shader language and as such is executed solely on the GPU as desired. The CPU handles the window management and the loading of the vertex data which is then passed onto the GPU in various data structures. An interesting thing to note about this, whilst a position in three-dimensional space requires only 3 values to be represented, the data sent to the shader consists of 4 values, although the 4th value is not used in the calculations. This is because I discovered an issue with sending only 3 values to the shader in that the shader would read the values as a set of 4, regardless of the actual size sent. This meant it would then read into the next set of values as they are laid out in memory linearly. This means if there were less values than expected then the shader would take the value from the next set, skewing the expected data structure. This is why all data sent to the shader consists of 4 values.

Initial Cube Implementation

For my initial implementation I used a simple ray-cube intersection algorithm in order to get familiar with the rendering techniques of raycasting and to better understand how the OpenGL compute shaders worked. I found a tutorial with code examples on GitHub for a Java implementation of a simple raycasting algorithm using compute shaders. Using this I was able to translate and expand the code from Java into C++. This algorithm worked well for low numbers of cubes but since the only objects that were able to be rendered were cubes, it did not make for very interesting scenes. Also with no kind of filtering or proper data structure, there was an attempt to render every cube each frame, regardless of whether it was actually visible or not. So the program's frame rate decreased rapidly as more cubes were added.

Below is the algorithm used to determine whether for a specific ray there was an intersection:

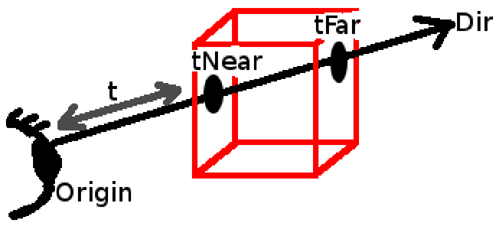
```
bool intersectCubes(vec3 origin, vec3 dir, out hitinfo info)
 {
   float smallest = MAX_SCENE_BOUNDS;
   bool found = false;
   for (int i = 0; i < NUM_CUBES; i++)
   {
     vec2 lambda = intersectCube(origin, dir, data[i]);
     if (lambda.x > 0.0 && lambda.x < lambda.y && lambda.x < smallest)
```

```
        {
            info.lambda = lambda;
            info.bi = i;
            info.cubeMin = data[i].min;
            info.cubeMax = data[i].max;
            smallest = lambda.x;
            found = true;
        }
    }
    return found;
  }
```

This code loops through all the cube data that was sent to the GPU and then computes whether, for a given ray, there is an intersection. The intersection code returns a 2 dimensional vector containing the far intersection point (the point the ray leaves the box) and the near intersection point. If the near intersection point is smaller than the far point and is smaller than the current nearest point then we have a new closer intersection. Below is the individual cube intersection code:

```
vec2 intersectCube(vec3 origin, vec3 dir, const cube c)
{
  vec3 tMin = (c.min - origin) / dir;
  vec3 tMax = (c.max - origin) / dir;
  vec3 t1 = min(tMin, tMax);
  vec3 t2 = max(tMin, tMax);
  float tNear = max(max(t1.x, t1.y), t1.z);
  float tFar = min(min(t2.x, t2.y), t2.z);
  return vec2(tNear, tFar);
}
```

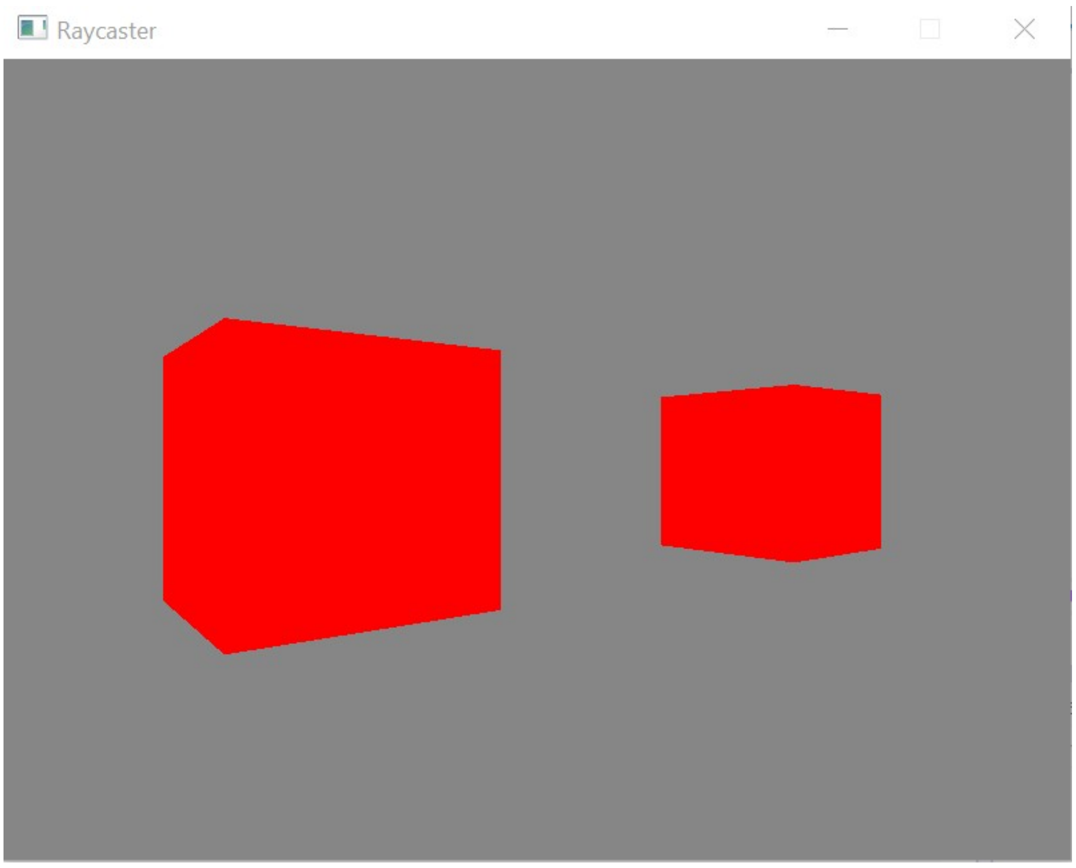
Using the parametric form of the ray (**origin + t*dir**) we work out the value **t** at which the ray enters the cube. We return this along with the far intersection point (the point at which the ray leaves the cube). If the ray doesn't hit the cube then the far point will be less than the near point, additionally if the near point is negative then the cube is behind the ray.



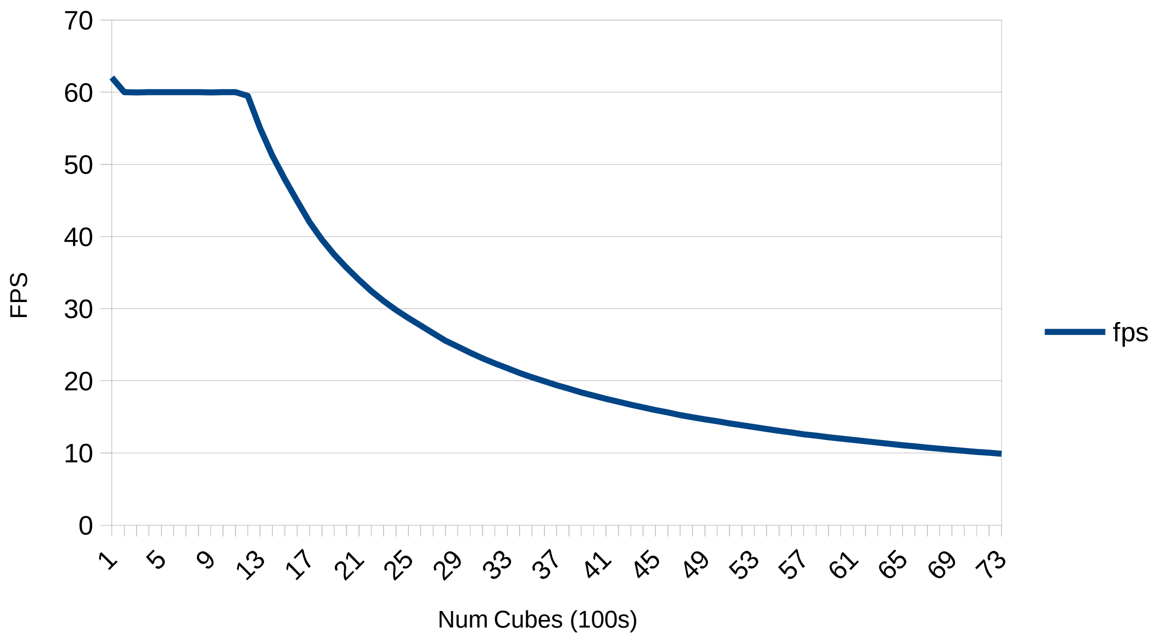
The scene for the cubes was hard coded on the CPU and the data was then sent to the GPU for rendering. The data structure for the cubes is very small and as such it does not require much memory to create and store a number of cubes for rendering.

```
struct cube {
    glm::vec4 cubeMin;
    glm::vec4 cubeMax;
};
```

Below is a picture of an example scene using this method:



The performance for the cubes was incredibly good. I was able to render up to 1500 cubes before the performance started to drop below 60 frames per second. As you can see by the graph below the maximum number of cubes I was able to render before hitting lower than 10 FPS was 7400. At this FPS the application is sluggish and movement around the scene is difficult due to input lag. It's still possible to change the view but it is a lot more unresponsive.



This is clearly many times more than the maximum number of cubes we would likely need to render, but the performance will drop as we add more and more features, so it is advantageous that we are already getting good results.

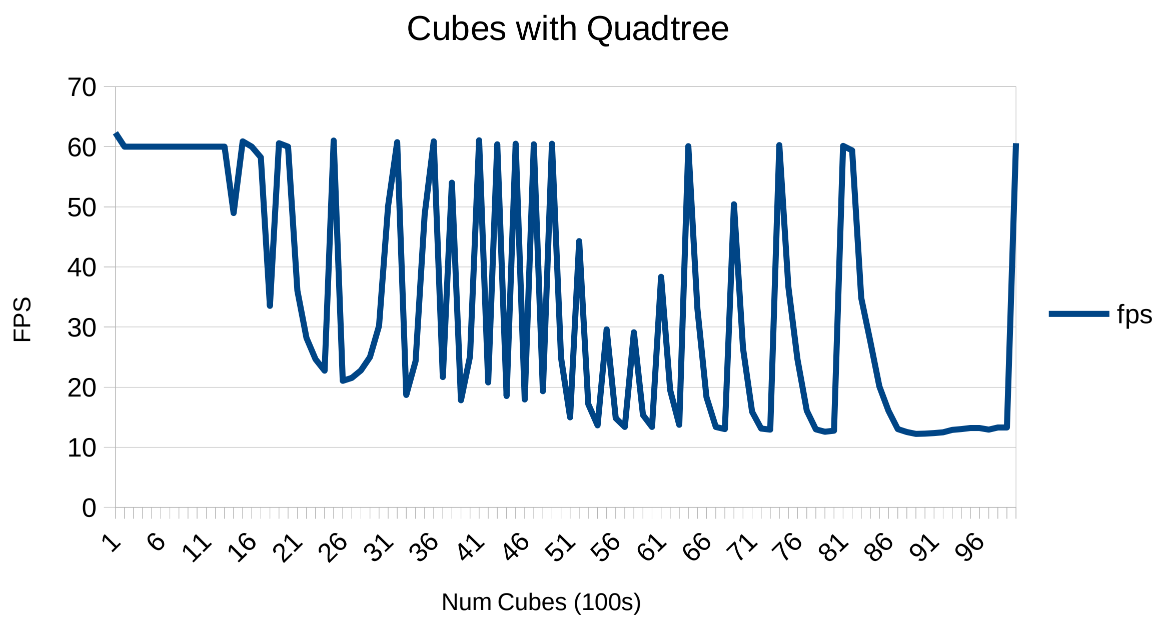
Quadtree Implementation

My next step was to improve this implementation by adding the cube data into a suitable structure like a Quadtree or an Octree to better decide what data should be sent to the graphics card for rendering. To start with I decided to, for simplicity's sake, implement a Quadtree data structure for the cubes and extend this to an Octree structure later if time allowed. I created a Quadtree class from scratch and implemented the core functions necessary for the system to work.

When utilising the Quadtree data structure the program did indeed see a performance boost as expected, although there was still a hard limit on the number of cubes able to be rendered, dependant on the resolution. Rather than continue to optimise further and expand my Quadtree structure to start using Octrees for object culling, I decided to move onto making the scene a little more interesting than just a couple of cubes.

The actual raycasting was not changed during this revision of the application. I did not have to change any of the ray intersection code, the only change was to which cubes I was actually sending to the GPU.

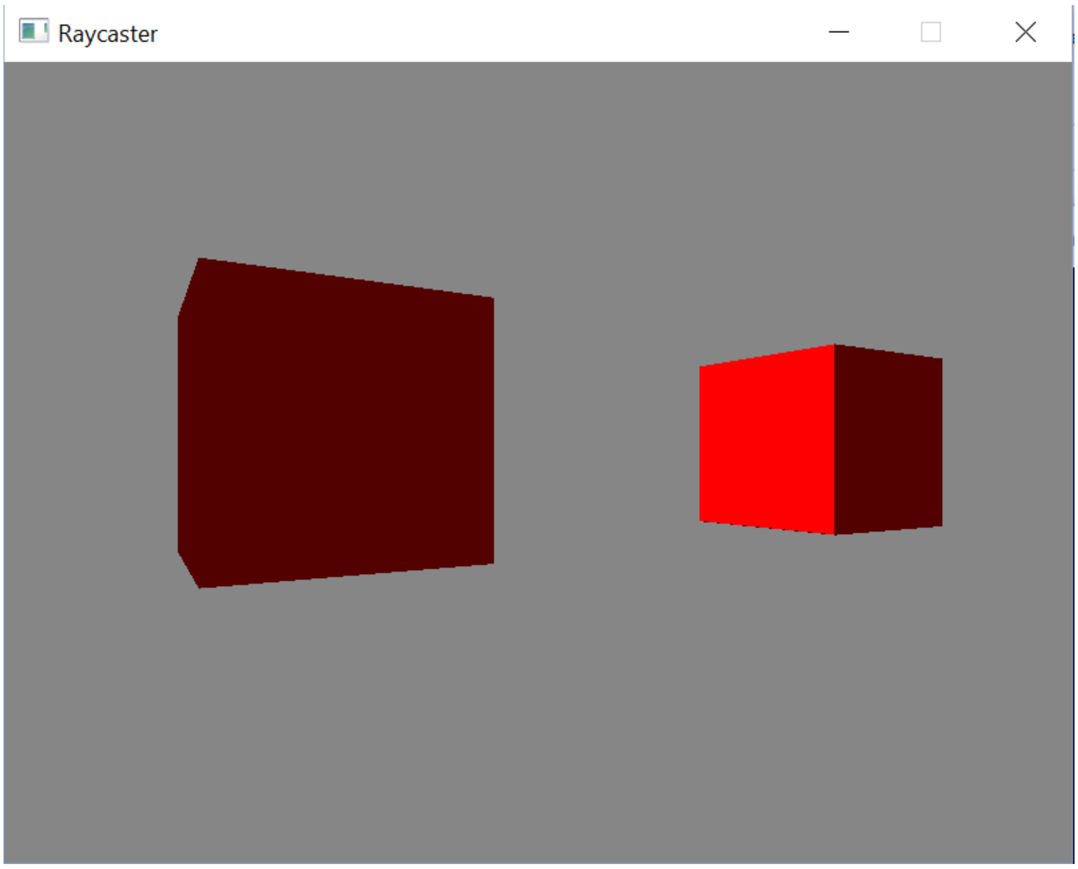
Below is the graph for the performance of the application when using the Quadtree data structure:



As you can see from the graph the application was able to maintain high frames per second with a large number of cubes. Although there were frequent drops in frames per second it is quite likely this is due to my naive implementation of the Quadtree class, however I did not consider this relevant to the project as I was investigating whether a Quadtree data structure would increase performance, and it clearly did. Given more time I could perfect this data structure and achieve more stable FPS, but once I had seen that my applications performance was able to be increased by utilising such data structures, I moved on to adding more features.

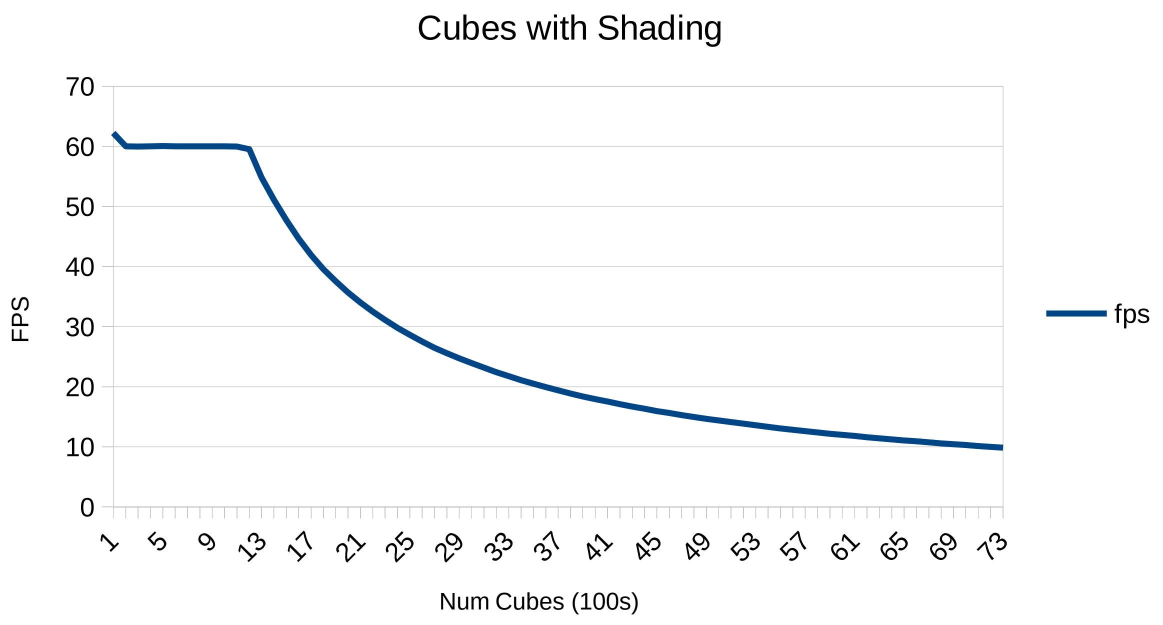
Simple Shading

Once the program had been sped up a little bit I decided to utilise some of the performance gains and implement simple shading. Since I was only going for straightforward shading I used Phong shading techniques to achieve simple yet effective shading. I chose the Phong shading technique as it is extremely lightweight and is one of the shading techniques I am most familiar with. However the technique does require the normals, therefore I had to slightly alter the ray-cube intersection code so that I could calculate the normal of the intersection point. This was fairly trivial however and once the shading was implemented there was no significant performance decrease. The scene was instantly much more appealing however. This can be seen below in the picture of 2 cubes with a light source in between them.



Since we are currently rendering only cubes the normals are easy to calculate. However were we to render more complicated geometry we would have to calculate these normals for each object. A way to overcome this and potentially another area we could optimise would be to calculate the normals on the CPU for each object, only once, and then send these to the GPU along with the other vertex data. This would be a very good optimisation as instead of calculating the normals each frame on the GPU we only have to do it once on the CPU. We can even calculate these offline so it doesn't affect any rendering or start-up time, not that the normal calculations should by themselves cause a performance decrease.

Below is a graph of the performance when shading was being used (without also using the Quadtree data structure as I just wanted to test performance against the cubes without optimisation):



When comparing the performance with shading against the performance without shading the difference is negligible. We were able to again render up to 7400 cubes before dropping below 10 FPS, clearly showing there was very little measurable impact of including a simple shading model in the application. Yet there was a significant increase in the appeal of the images rendered, this makes shading an inexpensive yet effective technique.

Triangle Intersection

With the simple shading for the cubes implemented I wanted to render more complex geometry. I had the option of either rendering more complex geometric shapes, or implementing ray-triangle intersection so that external model files could be loaded and rendered. The advantage of adding more ray intersections for complex shapes is that I could render more interesting geometry, however the scenes would still consist of uniform objects and easily become repetitive. If I implemented ray-triangle intersection then I could render any complex shape I wished with ease, increasing the variety of the objects I can render. The downside to this however is that this would be more computationally expensive as a whole.

For example, to achieve the ray-cube intersection it was only necessary to perform a few calculations for the whole cube. If we were to use ray-triangle intersections however, then we would need to compute 2 intersections per face of the cube, making for a total of 12 ray-triangle intersections needed to be computed per cube. Even if the ray-triangle intersection code was more efficient than the ray-cube intersection code, which it is not, this would be a significantly more calculations to render a cube.

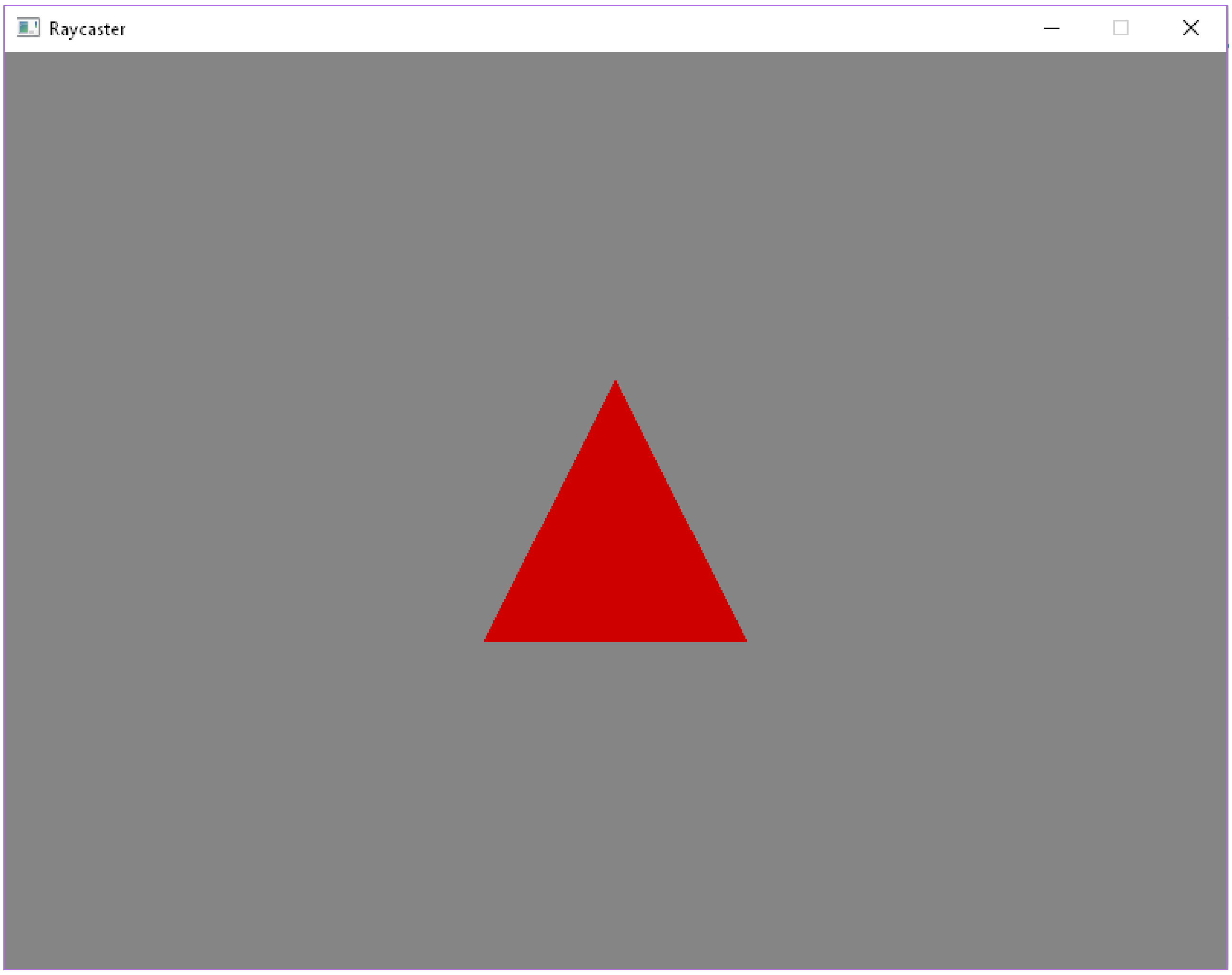
Despite requiring more intersections to create simple objects, one can render much more interesting models from external sources using this. For this reason I decided to implement the ray-triangle intersection code rather than implement specific geometric intersections.

I went through several iterations of this algorithm, each getting slowly more complex. For the first iteration I was just aiming to render a triangle, regardless of efficiency. To this end I used the algorithm below:

```
bool intersectTri(vec3 origin, vec3 dir, const Tri tri)
{
    vec3 v0v1 = tri.v1 - tri.v0;
    vec3 v0v2 = tri.v2 - tri.v0;

    vec3 N = cross(v0v1,v0v2);

    float NdotRayDirection = dot(N,dir);
    if (abs(NdotRayDirection) < 1e-8) // almost 0
        return false; // they are parallel so they don't intersect !

    float d = dot(N,tri.v0);

    float t = (dot(N,origin) + d) / NdotRayDirection;
    // check if the triangle is in behind the ray
    if (t < 0) return false; // the triangle is behind

    vec3 P = origin + t * dir;

    vec3 C; // vector perpendicular to triangle's plane

    // edge 0
    vec3 edge0 = tri.v1 - tri.v0;
    vec3 vp0 = P - tri.v0;
    C = cross(edge0,vp0);
    if (dot(N,C) < 0) return false; // P is on the right side

    // edge 1
    vec3 edge1 = tri.v2 - tri.v1;
    vec3 vp1 = P - tri.v1;
    C = cross(edge1,vp1);
    if (dot(N,C) < 0)  return false; // P is on the right side

    // edge 2
    vec3 edge2 = tri.v0 - tri.v2;
    vec3 vp2 = P - tri.v2;
    C = cross(edge2,vp2);
    if (dot(N,C) < 0) return false; // P is on the right side;

    return true;
}
```

This algorithm starts out by using the triangle vertices to produce vectors **v0v1** and **v0v2**. From these we can calculate the normal by finding the cross product between them. Once we have the normal of the triangle we can compute the dot product between the normal and the ray direction to find out whether the triangle is parallel to the ray, and thus whether there will ever be an intersection. We can then compute **t** which is used in the parametric equation (**origin+t*dir**) to compute the intersection point. If **t** is negative then the triangle is behind the ray and will never be hit, so we can exit without further calculations. Otherwise we can calculate the intersection point **P** using the parametric equation. We now need to determine if **P** is inside the triangle or not. To do this we use the “inside-outside” test. This involves calculating cross product between an edge and the vector between point **P** and one of the vertices involved in the edge. If **P** is indeed inside the triangle then the cross product between these two vectors should be in the same direction as the normal. To find out if the cross product is indeed the same direction as the normal we find the dot product between the two, if this is less than 0 then it is in the opposite direction to the normal, and thus the point **P** lies outside the triangle. We then just perform the ‘inside-outside’ test on all three edges of the triangle to work out whether **P** is inside the triangle or not. If **P** is indeed inside the triangle then we have hit a pixel within it and we can pass this information on in the shader to determine what colour to set the pixel as.

This worked and produced the result seen below:



While this method worked there were several problems with it. There were a lot of unnecessary calculations being made, such as the calculation for the normal of the triangle. It’s easy to get this information and pass it to the GPU rather than calculate it every frame. Second there was no way to interpolate the vertices to get exactly the point of the triangle hit, as the algorithm didn't make use of Barycentric co-ordinates. As I was researching other ray triangle intersect algorithms to solve these issues, I came across the most cited and most widely used example, the Möller–Trumbore (Möller and Trumbore 1997) triangle intersection algorithm. In order to make the ray-triangle intersection algorithm more efficient I implemented this, it also has the benefit of having Barycentric co-ordinates.

Below is the code for this:

```
float intersectTri(vec3 origin, vec3 dir, const Tri tri, out vec2 tex)
{
    vec3 v0v1 = tri.v1 - tri.v0;
    vec3 v0v2 = tri.v2 - tri.v0;
    vec3 pvec = cross(dir, v0v2);
    float det = dot(v0v1, pvec);

    if(abs(det) < 1e-8) //Triangle is parallel
    {
        return -1;
    }

    float invDet = 1/det;

    vec3 tvec = origin - tri.v0;
    float u = dot(tvec, pvec) * invDet;
    if(u < 0 || u > 1)
    {
        return -1;
    }

    vec3 qvec = cross(tvec, v0v1);
    float v = dot(dir, qvec) * invDet;
    if(v < 0 || u + v > 1)
    {
        return -1;
    }

    //If the texture co-ords are less than 0
    //then there is no texture information
    if(tri.tex0.x > 0)
    {
        vec3 temp = u*tri.tex0 + v*tri.tex1 + (1-u-v)*tri.tex2;
        tex.x = temp.x;
        tex.y = temp.y;
    }
    else
    {
        tex.x = tri.tex0.x;
        tex.y = tri.tex0.y;
    }

    float t = dot(v0v2, qvec) * invDet;

    return t;
}
```

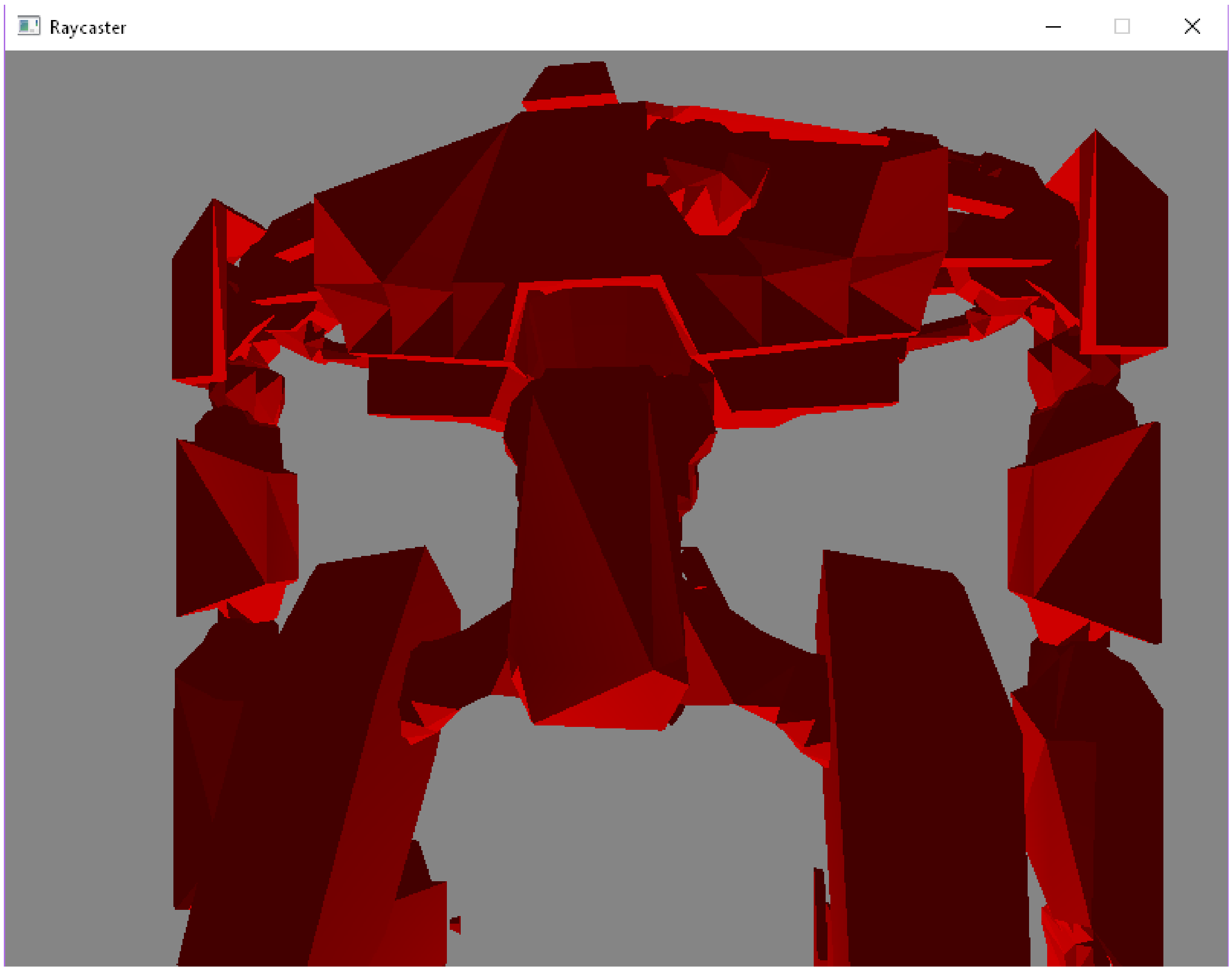
The Möller-Trumbore algorithm is a lot more complicated than the previous triangle intersection so I won't go into detail about how it works, that information can be found in the paper I have referenced. The tests for whether we hit the triangle however are very similar, even if the ways used to get the values are different. We still test whether the triangle is parallel to the ray and whether the point **P** is inside the triangle or not using the dot product between an edge and another vector. The method to test whether the point **P** is inside or not uses Barycentric co-ordinates and their properties, rather than the "inside-outside" test we previously used however. We use the property of the Barycentric co-ordinates that **U+V+W = 1** and the fact that each variable must be between 0 and 1. Therefore we can test whether the triangle was hit by evaluating whether these conditions are satisfied. Finally if we evaluate that **P** is indeed inside the triangle and was hit by the ray, we return the value **t** so that we can calculate **P** later on if needed, and see how far away the triangle that was hit is. This information is used to see if we have hit a triangle that is closer to us than previously was hit. This allows us to ensure we don't "see through" objects as we render the closest object to us.

In order to fully test the Möller-Trumbore algorithm and whether it's actually more efficient I needed a lot more triangles to attempt to render. The best way of achieving this was to implement model loading and then rendering the model using each triangle intersection algorithm.

Model Loading

Now that ray-triangle intersections were implemented I could render any object, provided I had the necessary vertex information. To load object models I used the third party library ASSIMP. This library loaded all the vertex information I needed into it's own data structure, I then needed to parse this and convert it into my own data structure that would be then passed to the shader.

This was extremely straightforward, especially as the library had a flag to ensure all polygons in the object were imported as triangles, something I obviously would need since I only had ray-triangle intersections. Once the data was converted to my format it was just a simple case of passing the data to the shader and rendering the triangles:



The model pictured above contains 4467 triangles. Using my initial algorithm I was able to render this object at a framerate of 142.67ms of drawing time between frames (7 frames per second). After implementing the Möller–Trumbore algorithm I was able to render the model at a framerate of 136.98ms of drawing time between frames (7 frames per second). Despite having the same frames per second, there was still a fairly significant decrease of 8ms between frames, something that can clearly be optimised further but is a good starting point and it clearly shows that the Möller–Trumbore is indeed more efficient.

Textures

Now that the models were able to be rendered I wanted to load their textures and provide a more interesting scene than just block colours, albeit with shading. This is where the third party library SOIL came in. SOIL made it extremely easy to load images and pass the pixel data to the shader where it could be used to calculate the texture pixel that was hit in the object. Thankfully ASSIMP also loaded texture information, so all I needed to change was my own data structure so that I could pass this information to the shader as well.

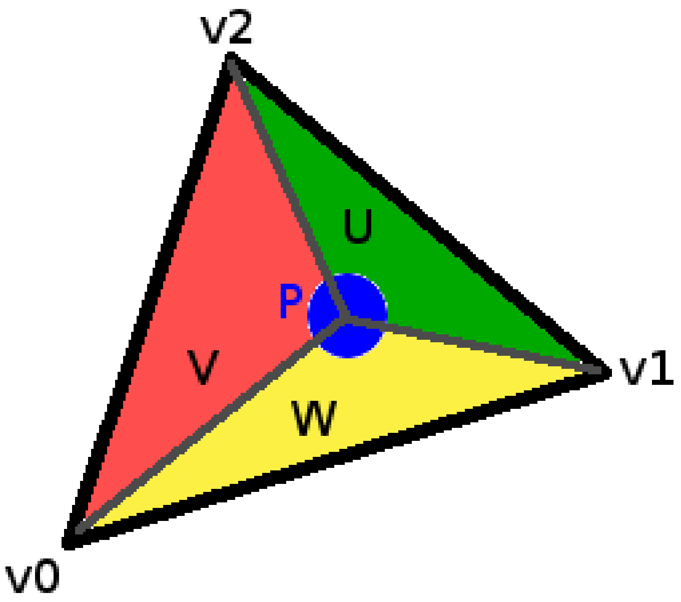
The triangle data structure looked like this before the addition of texture co-ordinates:

```
struct Tri {
 vec3 v0;
 vec3 v1;
 vec3 v2;
 vec3 norm;
}
```

And after the addition of texture co-ordinates:

```
struct Tri {
 vec3 v0;
 vec3 v1;
 vec3 v2;
 vec3 norm;
 vec3 tex0;
 vec3 tex1;
 vec3 tex2;
}
```

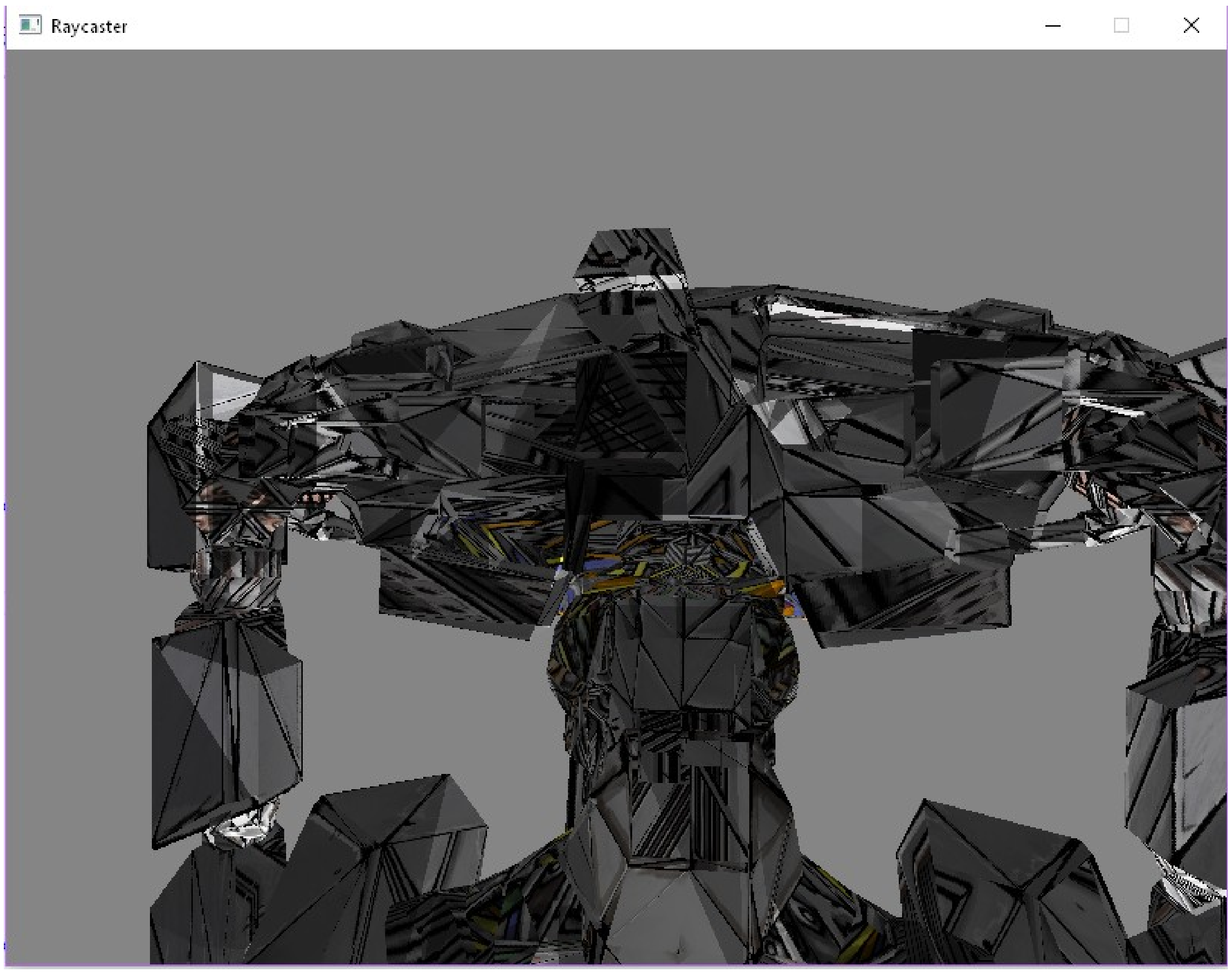
As you can see this is almost twice as much data being sent to the GPU, but it is necessary for creating interesting scenes. Since each vertex had a texture unit associated with it I needed 3 texture co-ordinates per triangle. A problem here is that the texture co-ordinate is known should the ray hit one of the triangle vertices directly; however if the ray hit anywhere within the triangle I would have to interpolate these texture units to get the correct texture pixel that was hit.



The Möller–Trumbore algorithm has this covered already using the previously mentioned Barycentric co-ordinates. It's worth noting that since **U+V+W = 1** (since Barycentric co-ordinates are normalised) we can replace **W** with **1-U-V** and then we only need to know the two variables **U** and **V**. The distances to the hit point are recorded in the variables **U** and **V** in the algorithm. With these it's just a simple matter of using the following equation to get the interpolated hit point.

$$P = uA + vB + wC.$$

This then produces the desired result:



EVALUATION

Despite running out of time to implement some of my extra goals, such as the extra effects like reflections, I reached my goal of creating an application that could render in real time a complex model with textures and simple shading.

The performance of the application depended heavily on the complexity of the external model attempting to be rendered. As shown above, I was able to render thousands of cubes before the performance dropped to a point where moving around in the scene become problematic and the application became more unresponsive. When loading external models however I was unable to render the same number of triangles as I had cubes, due to the additional complexity of calculating ray intersections.

When loading a model with approximately 200 triangles the application had a performance of 16.6ms per frame. This translates to approximately 60 frames per second, an extremely good result considering that most real-time applications try to hit 60 frames per second, and even in some cases 30 frames per second is all that is needed.

As soon as the number of triangles is increased however the performance drops, as expected of course. At around 1000 triangles the application achieved a framerate of 28.7ms per frame, roughly 35 frames per second. A significant drop, although this is still a perfectly acceptable framerate.

At around 4000 triangles the application starts to become unresponsive as the framerate is 122.5ms per frame, which translates to roughly 8 frames per second. At these speeds the application feels very sluggish, but it is still possible to move around an observe the scene, albeit slowly.

The application falls down at about 18000 triangles and becomes highly unresponsive. It takes several seconds to register input and movement around the scene is almost impossible. The framerate at this number of triangles was 576.6ms per frame, almost 2 frames per second.

It’s worth noting that these are static scenes without any sort of animation going on. Although you are able to move the camera freely in the scene, the models themselves do not move or animate in any such way. As such the performance is much higher than would be otherwise. Along with this, when textures are loaded, only one texture is used per model so there are not multiple large images being loaded into the application. Were the application changed to support multiple textures then I predict that there would be another performance drop as more memory is allocated to handle these potentially huge image stores.

The rendering of these models was very costly, but my Quadtree implementation didn’t work correctly for the loaded object models as each triangle was so close together, so I was unable to find any performance gains here. Given further development time, this Quadtree implementation would have been corrected, or a modification made to use a third party library in order to correct this.

Unfortunately this might not have even sped up the application so much, at least when one object was on screen, just because of the number of triangles contained in such a small space. A Quadtree wouldn’t cull these triangles and so the data structure would have no effect. A different way of optimising this would need to be found, such as using a three dimensional version of the Quadtree data structure like an Octree or another method entirely of culling objects.

Unfortunately I was not able to achieve my extra goal of including some ray tracing effects such as reflection and despite not managing to optimise the application any further, I had achieved my goal of implementing the basics of a raycasting rendering engine. On top of this I also met the goal of being able to render complex objects at a good framerate.

FUTURE IMPROVEMENTS

There are several areas I could improve on in this project were I to have more time. I would add some extra features such as the ability to load more than one model and load more than one texture per model. I would also try to add some more shader effects such as reflections or fog. Along with these effects I would improve on the currently very simple shading model, adding support for things such as multiple lights.

One of the major things I would like to do is optimise the application further. I'm very pleased with how the application performs, but I would like to be able to render even more complex models. A way to achieve this would be to re-implement a sorting data structure so that it's possible to minimise the number of triangles being tested for ray intersections each frame. It would be best to try to avoid sorting these triangles every frame as that would also have an impact on performance, so a system would need to be implemented that would allow for this.

During my development of the project, the next generation OpenGL API was released, Vulkan. It is a low-overhead, cross platform 3D graphics and compute API designed to let developers have even more control over the GPU and have lower CPU usage. It was also designed to better distribute work over multiple CPU cores, something that would be extremely beneficial to a high performance application such as this. One of the major benefits of this new API is the unified management of compute kernels and graphical shaders, eliminating the need to use a separate compute API in conjunction with a graphics API. This makes Vulkan the perfect API for developing a real time raycasting engine. Were I to do this project over again I would spend my time familiarising myself with the new Vulkan API so that I could utilise this in my project. The downside of Vulkan is that it's extremely verbose, as it gives the developer so much control, that it would take me significantly more time to develop the application. However the results I believe would be much better, and more optimisations and improvements could be made to the application.

REFLECTIONS

Over the course of this project I have developed my skills in a variety of areas and learned a whole lot more about the optimisation and graphics than I had previously learned in lectures or my own time.

At the beginning of the project I knew only a small amount about what raycasting was and how it worked. I spent the first couple of weeks at the beginning of my project researching the raycasting technique and how it might be implemented. During this time I learned much about how rendering is achieved at a lower level and found lots of resources about how one could actually go about doing it. As soon as I had enough information I dived in and started on my project, even though my knowledge of the OpenGL API was fairly limited. This soon changed as working on the project had the effect of increasing my knowledge of the OpenGL API because I was forced to use the bleeding edge of the API and had to dig deep into the code to get what I needed. Thanks to this I also better understand how the OpenGL rendering actually works. My theory of graphics has improved considerably too due to all the research I had conducted into how raycasting works verses the other methods such as rasterisation. On top of this, because of all of the mathematics within such algorithms, my knowledge of vectors, geometry, general algebra and calculus has been strengthened.

Optimisation has been a huge area of improvement for me as well. Before starting this project I had never made a conscious effort to optimise my software, as I had never needed to because I wasn't trying to develop high performance applications. After finishing this project I'm now much more aware of how to optimise software and in what places it is necessary. This project has given me a huge insight into how high performance applications are developed and has also introduced me to a variety of data structures designed to help the optimisation of such software.

Overall I am extremely happy with what I have achieved and what I have learnt during the course of this project. Before starting I already had a deep interest in graphics, and now that I've finished the project I have an even greater interest in this area, specifically at a low level. I am eager to continue my research into graphics and see what else I can learn on this topic. I look forward to continuing my development on this project in my own time, implementing all of the things I had planned and maybe re-writing the project to make use of the new technologies being released.

CONCLUSION

I started the project with the intention of creating a simple real-time raycasting rendering engine and I believe I succeeded. I started out by being able to render simple geometry and kept expanding on this, adding support for shading and then trying to optimise the application by utilising data structures to cull out of view objects. I added support for loading external models to allow for more complex geometry and then extended this support to allow textures to be loaded as well.
This brought me to the end result in which I have managed to create an application that can load external complex models, not just simple geometric shapes, and can render these in real-time.

This doesn’t mean the project is finished for me however. I am going to continue to work on this and see if I am able to render numerous complex models with multiple textures and then, once I am more familiar with the new API, I will re-write the application in Vulkan to see if I can further improve the performance using the latest technologies.

References

Hanan Samet (2006). *Foundations of Multidimensional and Metric Data Structures.* Burlington, Massachusetts: Morgan Kaufmann Publishers Inc.

Ioannis Tsakpinis (2014) *Ray-tracing with OpenGL Compute Shaders*
Available at: https://github.com/LWJGL/lwjgl3-wiki/wiki/2.6.1.-Ray-tracing-with-OpenGL-Compute-Shaders-(Part-I)

Möller and Trumbore (1997) *Fast, Minimum Storage Ray/Triangle Intersection*
Available at:
http://www.cs.virginia.edu/~gfx/Courses/2003/ImageSynthesis/papers/Acceleration/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

OTOY (2014) *Brigade Engine*
Available at: https://home.otoy.com/render/brigade/

Scratchapixel (2011) *Ray tracing – Rendering a triangle*
Available at: http://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle

External Libraries Used

ASSIMP: http://www.assimp.org/
GLEW: http://glew.sourceforge.net/
GLFW: http://www.glfw.org/
GLM: http://glm.g-truc.net/0.9.7/index.html
SOIL: http://www.lonesock.net/soil.html

GitHub Project Repository

https://github.com/SimonRhys/RayCaster