**AIS Report**

**Jonathan (Noel) Wilson**

**F6027955**

**Contents**

**Introduction** page 3

**Research** page 3

**Analysis and design** page 4

**Implementation and design** page 5

**Conclusion** page 8

**Introduction**

For this project I have several aims that I wish to achieve, firstly I aimed to build a basic ray tracing renderer which has the ability to fire rays into a scene and return a colour. After this was completed , set-up my renderer to bounce these rays around the scene in order to trace shadows, reflections and refractions of objects. Another goal for this renderer was to import models from another modelling package, then create advanced modelling systems such as subdivision which would allow my renderer to render complex shapes. Finally I would build on my ray tracer to turn it into a photon mapping ray tracer.

**Research**

The majority of my research came from the book “Physically Based Rendering” by Matt Phar and Greg Humphreys. This book provided me with the majority of the knowledge required to build and improve my renderer. The rest of my research came from our tutor who was able to provide us with notes and diagrams about ray tracing renderers.

The theory for the ray tracer is from notes given to us by Dr Mathew Holton - University of Teesside. These notes gave me the base knowledge of the ray tracer, the viewing system with active and passive cameras and knowledge of how to build a whitted ray tracer, with rays being cast through the camera into a scene then being reflected or refracted once around the scene.

The Ray system which is used to manipulate and control rays in the renderer is desgined based on the system by “Physically Based Rendering” page57. Using it’s various methods for calculating intersections, with different objects including the methods for calculating interactions with implicit shapes to speed up my renderer.

The way I chose to store and build my models was creating using the methods from “Physically Based Rendering” page 139. This method Consisted of storing all models as lists of positions, normals and UVs. Each face was then stored in a Face object which held pointers to each of these three piece of data for each vertex.

Loops subdivision technique was used and I am using the methods and algorithm that are described in “Physically Based Rendering” page 133 “3.7 subdivision surfaces”, this seemed the most versatile solution being able to be used on every mesh as it operated on triangles. This meant it was a superior choice to Catmull and Cark subdivision which focused on quads meaning that each model would be required to be converted to this format increasing the workload and complexity required, while achieving no better a result.

I read into radiometry from “Physically Based Rendering” starting page 237, how each light in the scene emits energy into the scene based on a percentage of it’s total area divided by the area of the object hit projected onto the light. I then discussed more efficient methods of doing this than in the book with Dr Mathew Holton.

I was also advised to research photon mapping as this suited my renderer much more than radiometry, firstly discussing it with Dr Holton then researching it myself. I read into “Physically Based Rendering” from page 769 and considered how I would firstly modify my rays to build a new photon class starting from a simple ray with a weight and flux value. Consideration was also put into how I would store each hit in a map structure that would link the hits to objects making getting the hits a process involving, find thing the model the ray has hit, then looking through the maps for that model and listing it’s photons, finally averaging them up by a sum of the values modified by weights applied due to their proximity.

Sadly I was unable to attempt to implement these as I had problems with my modelling system.

**Analysis and Design**

The renderer is designed in a fairly simple way which concentrates on storing and using models efficiently as it’s major focus was to incorporate a complex modelling system. This means that the data used to build any of the models is stored once and is if possible referenced or pointed to when required, a lot of consideration was taken into how to store sub-division models efficiently so that their levels could be ascended and descended quickly. This will add complexity but also efficiency as long as I make sure I continue so always make the pointers available what requires the data. This makes my renderer less robust but easier to program and faster, I had to be careful with this design so that I do not give access to the data where it is not required.

The design of the ray tracer was to create a witted ray tracer which could then be developed into a photon mapping system. It would have a control structure that would manage all the nodes in the renderer. This would mean that I would have to rewrite and redo some of the renderer, but I accept this as I will have to progress through this project as I learn more about ray tracing. The witted ray tracer is a simple concept and has the bonus of being easy to understand and easy to implement so it will give me a solid base to work on in this project. I plan to protect my base code with polymorphism (see below) so that I can work around problems that arise and solve them quicker. The photon mapping system is one of the best ways to represent the LTE in a renderer as it works out the energy in a scene and where it goes similar to radiometry but is more flexible, gives better results and is much more compatible with a whitted ray tracer. Using similar techniques but with much more advanced calculations, as with radiometry I would have to rewrite my renderer as it uses completely different techniques.

Polymorphism is planned to be used in this project a lot to enable multiple type of objects be stored and handled at the same time with the ability to expand easily. This technique will allow me to create and easily add functionality or extend my renderer, it will be very useful for multiple objects such as lights, which can have several type stored in the one container and all processed at the same time. With the ability to add more further into the project, it will also allow me to expand my modelling system, as I can have a basic model class and derive more complex types for different types of model which will also make my renderer more efficient as models will only have data types assigned to them if they are required.

For soft shadow the area light design was used, this was added through polymorphic methods and is an effective way of creating fading shadows, although it can be slow if assigned too many samples and produce artefacts if too few. A good balance is needed when using this technique to mix speed with quality.

Implicit shapes are a large part of the design in this renderer, these are more complicated to produce than regular shapes as they are stored as a formula to calculate their points instead of being actually geometry that is stored. This does mean that they are very fast to test rays against as they only require one test to pass or fail them. This means they are excellent to be used to pass or fail rays that will not touch a complex model at all, as they will fail all the surrounding rays quickly, meaning that complex calculations are only needed for rays touching the model or in close proximity to it. They are also a very accurate way of creating basic geometry which is very useful for testing in renderers.

**Implementation**

The renderer is controlled by a JWRT object which organises and controls all the other classes and functions. It firstly initialises the framebuffer and the scene either by loading files or by code. JWRT uses the model system which uses the model classes to build models to be put into the scene. JWRT then initialises all textures, materials, lights and implicit shapes in the scene. (Fig 1.0)

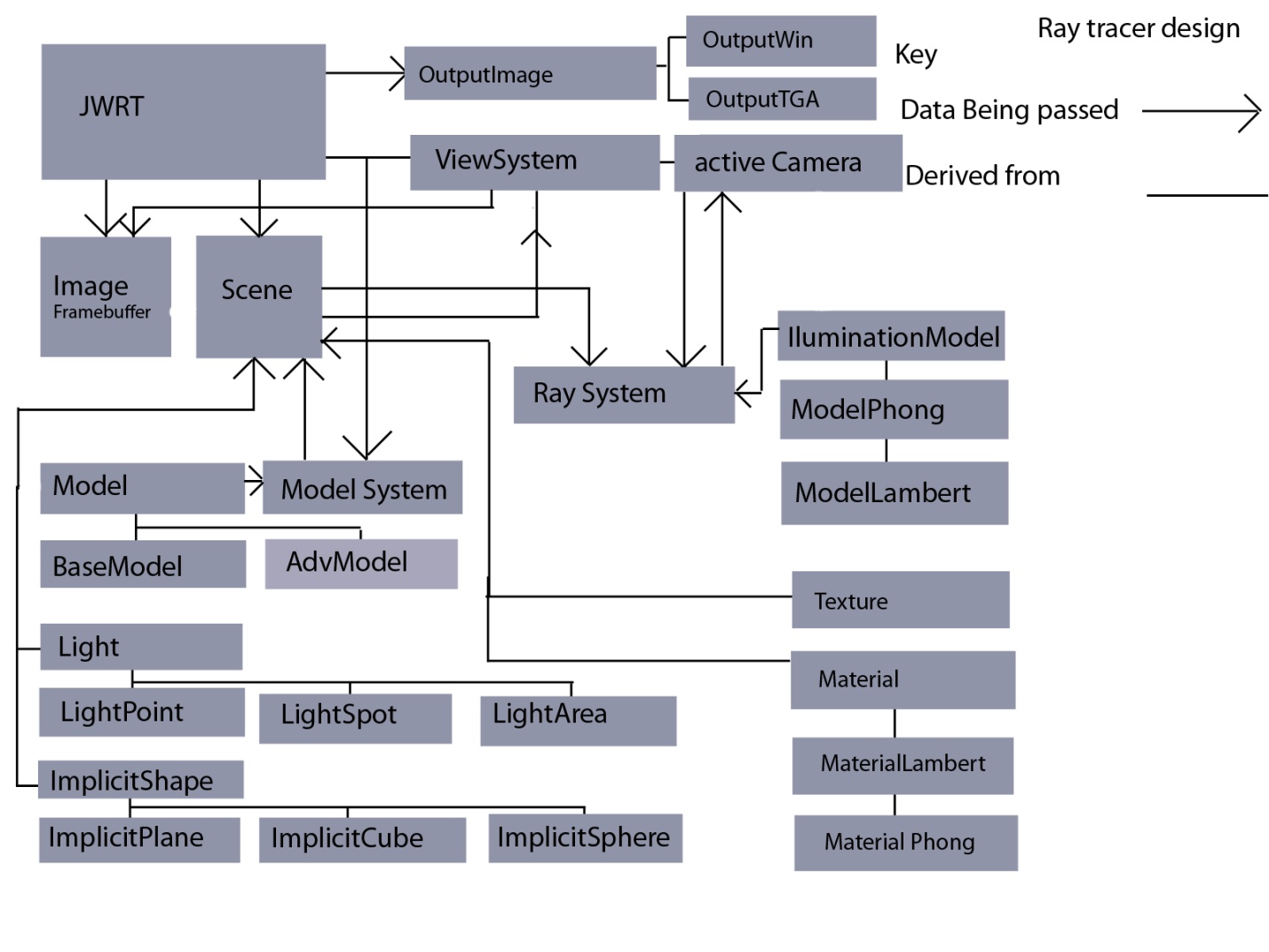


Fig 1.0

Once initialised JWRT will initialise the view system and select either the active or passive camera (only active shown) and pass it a pointer to the scene to render. The camera will pass this pointer to the ray system and using the scene data the veiwSystem and active Camera will fire rays into the scene using the ray system. The ray system then checks for collisions in the scene, once it has collisions it will get the model, material and texture data about the model it has struck and work out the return colour. The return colour is calculated by using the collision data, illumination model and the light data in the scene. Once the ray system has this colour it will check to see if it should do a “bounce” and if so will calculate new rays and fire them into the scene again, adding the colour these rays return to the sum of the colour for the pixel. Shadows are also done by the ray system which will use the scene data to work out if the collision point is obstructed or what percentage of it is obstructed if it is soft lighting.

One module not on diagram is the fileReader class which using it’s derived children can read in custom objects, the renderer can import and render .obj models with textures. JWRT creates an instance of fileReader, passes it a pointer to the scene then given the correct file name fileReader will read the file create it’s model then store it in scene.

All Data management is done by scene and JWRT, when the program closes scene deletes all dynamically created content and JWRT does the same with the camera, image and the other components. Other classes have access to scene and can edit it through pointers that are passed to them.

**Rays and Ray System and collisions**

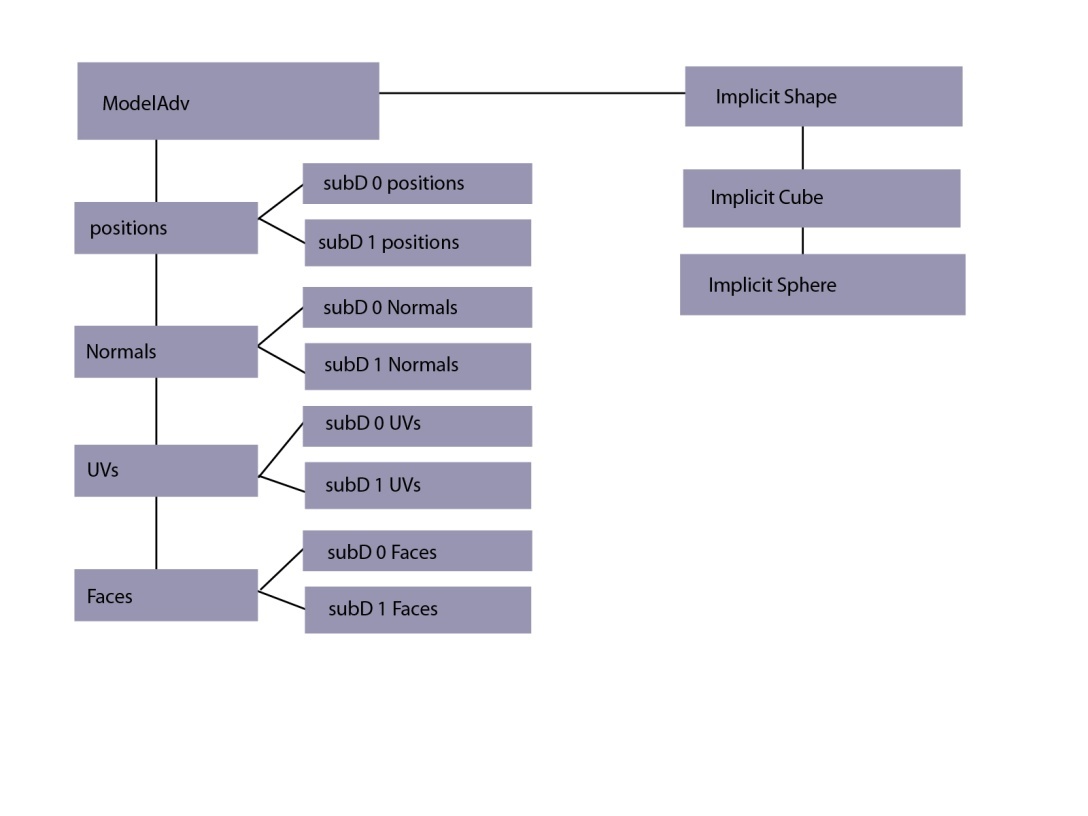
The Rays I used had an additional value which I choose to add at first for debugging purposes but it was left due to how easy it made passing data, the Ray class has a direction and a position, it also has a pointer which can be used to store a dynamically assigned vertex which can hold all the data required from a point of impact. This made passing this information very fast and easy but I did have add checks to make sure that the memory is cleared if a new point is required or if the point was no longer required. I believe I have done so in the renderer but I always had to bear this in mind while developing the render. I also chose to have a max and min value of T stored in the ray so that finding the closest intersection point to the camera would be easier. I saw this method in physically based rendering and realised it would work well with my renderer.

The ray system is a very power full class and does all the ray calculations in my renderer apart from intersections which it will pass the ray to the object struck to find the intersection. It can find all the refracting, refracting rays as well as handling all the shadow tests.

The actual collisions are handled by each different object type as they require different techniques to find their point of intersection. With polygon meshes this is done by splitting them up into triangles testing the triangles, but for implicit shapes each one requires a different technique for finding the intersection quickly and efficiently. The implicit shapes are much faster to use than polygonal meshes, so each model has an implicit shape pointer which can be used to wrap an implicit model around the mesh. This implicit shape is tested before the mesh speeding up rendering by quickly failing all the missed rays.

**Models and storage**

The model class was updated and changed heavily until it turned into the ModelAdv class which is used throughout the project, ModelBase was the original layout and although the render is built to use both, ModelBase is now obsolete and needs updating if I wish to use it again. The structure if ModelBase is so that no data is copied and so that it is efficient as possible. It holds a pointer to an implicit shape which can be used to hold any of the implicit objects so tests on the model can be performed quickly, this could be expanded to hold a vector of pointers so a model could be made of multiple explicit objects speeding up render time more. The class hold vectors of vectors holding positions, normals and UVs. (See Fig 2.0) Each one holds data for a different subD level on the model enabling the user to move between subdivision levels quickly. Each model has a subD value which is used to store which level of subdivision it is on, this is used in all the get functions. So if at subdivision level 2 the get functions return positions, normals etc. from the data vectors in position 2. I will however need to expand this to see if the memory versus the computation cost is worth it. I may delete smaller models and only store the higher levels if it is worth the memory to store them, but the current structure is very flexible and allows for many different possibilities. Such as using the current system to hold proxy meshes which could be used to control the actual mesh.



(Fig 2.0)

**Conclusion**

A lot was learned in creating this renderer, how to render out images and objects using new techniques and methods which all yielded interesting results. My renderer is incomplete and I believe that this is mostly due to over complication and bad time management which I have learned to avoid. On my last project, it was far too simple and this caused me major problems later on so I tried to plan in this project for expansion and to make it a lot more efficient. This created many problems though and in future I will create much simpler solutions before developing them into more refined ones. I will time my project better as well as I had large gaps between working on this project. This made getting back into the project and remembering all the details about it slow and tedious process slowing down my work flow.

I am unhappy with my renderer as I wanted to get subdivision surfaces working and although I have the structure methods to accomplish this, I ran out of time before I could find the problem that was causing it to fail. This was very demoralising as I had checked and found the positions and normals to be correct but the way the face’s are linked to the objects the last step in the process is incorrect.

I plan to use this experience and take it forward onto other projects, the lessons I have learned and the techniques gained are invaluable. I hope to finish this renderer in my own spare time.

**Appendix**

**Books:**

Physically Based Rendering - Matt Phar and Greg Humphreys.

**People:**

Dr Mathew Holton – University of Teesside.