King's College London

7CCSMGPR

Group Project

Group of Seven Report

Authors:

Number	Name	Email
1756850	Othman Alkhamra	othman.alkhamra@kcl.ac.uk
1739256	David Benicek	david.benicek@kcl.ac.uk
1770922	Aadam Bari	aadam.bari@kcl.ac.uk
1425704	Hitesh Mankani Vinod	hitesh.mankani_vinod@kcl.ac.uk
1755013	Timothy Obimma	timothy.obimma@kcl.ac.uk
1783087	Nagarjuna Vaddiraju	nagarjuna.vaddiraju@kcl.ac.uk

supervised by
Dr. Laurence Tratt

March 22, 2018

Contents

Chapter 1

Introduction

In real life, we have different objects such as spheres, cubes ... etc. What's mutual between them is that they are all geometry objects. However, those objects might appear in different ways, based on their environment. For example, a mirror sphere placed in an environment with many light sources will react in a different way than a metal sphere. And so, it is clear that different factors such as the material type, light sources and many other factors, will produce many different outputs.

Ray tracing is one of the techniques which helps in producing different images for different environment variables. It is widely used in producing films, video games, animations and many other areas. It can be applied on any object and not just geometry objects.

1.1 What is the Idea behind Ray Tracing?

Before discussing the idea behind ray tracing, and explaining its algorithm. Let's begin with listing the basic required elements to run the ray tracing algorithm. Those elements will be in one container which will be called **scene**. The elements of the scene are:

- **Object(s)**: a scene can contain one or many objects. Each object can be one of the previously mentioned objects types or any other object, which exists in real life. For example, an object could be a sphere, triangle, tree, car, building ... etc.
- **Light source(s)**: The algorithm of ray tracing itself doesn't require having a light source. However, a scene without any light source isn't very practical for ray tracing.
- **Image Plane**: referred to as **window frame** in our implementation. This plane has a set width and height and it is divided into small squares, where each square covers a number of pixels on the objects of the scene.
- **Eye or Camera:** In order to render the scene, it is mandatory to have an eye or camera, with a certain position and direction. The camera is used to determine the way of looking to the scene elements. From the eye or camera, view rays are produced which travel through the image plane and determine the way the object is rendered.

Figure 1.1 shows an example of a scene with the previously listed elements. Now, and after discussing the elements of the scene, it is the time to describe the algorithm of ray tracing. Despite the implementation not being trivial, the main idea is quite simple; in each scene there will an eye or camera, from that

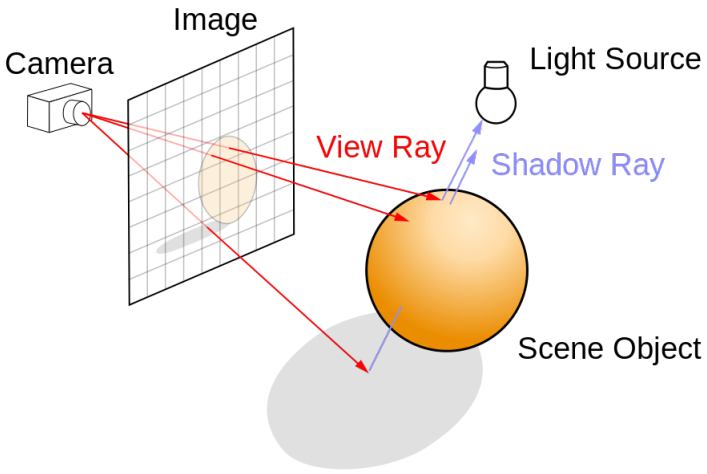

Figure 1.1: Ray Tracing's Scene Elements [1]

camera many rays will be produced and go across the small squares in the image plane, for each ray we need to check whether it intersects with any of the objects or not, and return the colour of that pixel. Other rays are possible to be produced caused by reflection. These rays also will follow the ray tracing algorithm before returning the final colour.

The algorithm itself is relatively simple. Nevertheless, as previously mentioned, environment variables play a big role in the calculations. As a result, returning the colour of the pixel, which was hit by the the ray, won't be that simple; since it will require additional computations related to the reflections, shades ... etc.

1.2 Our Solution

Over the course of the project we worked hard to deliver a final product that is able to render a scene created by the end-user using a web interface. The web interface provides both 2D and 3D views, where the user can create his own scene, and then just click render. After that, the user's input will be processed by the back-end engine, which in turn returns the rendered image as JPG file, with the name chosen by the end-user,back to the front-end.

The scene can contain multiple objects of three different supported objects. Each object can have any material from six different available materials in our program. In addition to that, the scene can have multiple light sources. Each of those elements will be discussed, in more details, later in **Chapter 4**.

Our solution showed great results for the inputs we defined, and in order to make our product more reliable, we covered both the front and the back ends with many tests. More details about the implementation and testing will be discussed later in **Chapter 4**.

Chapter 2

Review

2.1 Brief History of Ray tracing

Ray tracing is one of the most popular methods of rendering. What is Rendering? Rendering is the act of generating an image (usually photorealistic) from a 2D or 3D model. There are three familiar techniques to image rendering and they are rasterisation, ray casting and ray tracing. These three techniques are quite similar but are differentiated by their representation of the optical effects such as reflection intensity which in turn makes the processing speed vary differently (Wald, Slusallek, Benthin, & Wagner, 2001). This means that ray tracing would produce the most photorealistic image while also taking the most time to compute while rasterisation would produce the most basic image.

The earliest known Ray tracing system was calculated by hand by Richard Hoyt in the 1950's while working on vehicle shotline at the Ballistic Research Library known today as DARPA (Klopcic & L. Reed, 1999). In the 1960's, physicists coined the name "ray tracing" by plotting on paper the path taken by rays of light starting at a light source and passing through the lens in the design of lenses (Glassner, 1989).

Computer Graphic researchers at the university of Utah thought it would be a good idea to apply this simulation of light physics to create images but the computers of the 1960's were too slow to produce images of better quality than those made by using cheaper image rendering techniques hence the idea of ray tracing was dropped for several years.

Over several years, as computers grew more powerful, it became imperative to simulate the real physics. Several optimised algorithms were implemented and this led to computers being able to simulate various kinds of optical effects. Ray tracing have evolved greatly since then and is now one of the most powerful technique used in image rendering. Ray tracers have become both faster and more efficient and can even produce hyper-realistic images that are indistinguishable from the real world. We will Discuss some Ray tracers in 2.3.

2.2 Overview of Ray Tracing

Ray tracing is a simple and powerful technique. It involves shooting rays from a point-of-view (usually represented by an eye or a camera). These rays then go through each pixel of the viewing plane and some calculations are done to determine if they intersect with objects. When an intersection is detected, reflected or refracted rays are always generated (Weghorst, Hooper, & Greenberg, 1984). Each of the reflected/refracted rays must be recursively calculated to determine

which surfaces they intersect. At each pixel where intersection occurs, the ray tracer must work out the intensity of the light and how much of it reflected back in order to determine the exact colour of the pixel (Suffern, 2007). This is usually a long process and the time taken to generate each image increases exponentially with respect to an increase in the size of the viewing plane(image frame).

2.3 Review of Related Works

In this section, the Ray tracers discussed either helped influence our design decisions or have a similar structure to what we have built.

2.3.1 Ray Tracing from the Ground Up

This is a book written by Kevin Suffern.

This ray Tracer was developed using Object-Oriented(represented by OO henceforth) techniques due to its size and complexity. Ray tracers are incredibly large and complex and OO techniques are known for handling these. It also enables extensibility; Different types of object can be rendered without having to alter existing codes. This Book uses C++ as its development language because of its OO facilities and its computational efficiency. Some useful coding tips such as data storage, references and floating-point division were also gotten from this book. Important mathematical calculations such as sets, vectors, points, normal and the operations that can be carried out among them are also discussed extensively in this book.

The book focuses on the Ray Tracing engine (Equivalent to the back-end) and largely ignores any form of user-interface. The necessary inputs such as objects, material, light and frame size are hard-coded. The workings of this ray tracer can be represented using the pseudo-code below:

- Define some objects
- Specify material type for each object
- Define light sources
- Define a plane whose surface is covered with pixels
- For each pixel:

  Shoot a ray towards the object from the centre of the pixel

  Compute the nearest hit point of the ray with the objects (if any)

  - If the ray hits an object

    Use the objects material and the lights to calculate the pixel colour
  - Else

    Set the pixel colour to black

Colours are represented using their RGB value which is then stored in a 3D colour space. This allows various calculations to be carried out in order to produce an extensive variety of colours. See **Figure 2.1** below for representation of pure Red, Green and Blue.

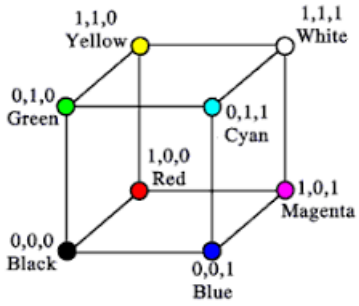

Figure 2.1: RGB colour scheme

This book also discusses the bounding-box technique. This technique is used to improve efficiency when an object is expensive(this refers to non-conventional shapes or objects such as animals or figures) to trace. This method only saves time if the bounding box is significantly cheaper to intersect than the object inside such as in **Figure 2.2** below. It works by enclosing the object in a rectangular box and when running the ray tracer, it first check if a ray misses the bounding box, then it cannot hit the object inside (Suffern, 2007). We did not implement this technique as our ray tracer only supports simple objects at this time.

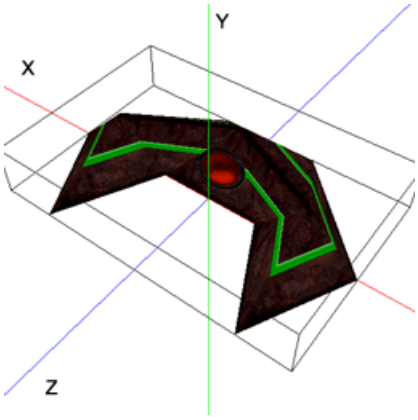

Figure 2.2: Bounding Box Example

Chapter 3

Requirements and design

3.1 Mandatory Requirements

The beginning of the project saw the group discuss and decide what features were essential for the running of the ray tracer, that is to say, creating and developing a running system, consisting of the front-end, which is the client side, and the back-end, which is the server side. The bare minimum requirements for the entire system were determined to be as follows:

- Allow users to create spheres and cubes with specific parameters' values. Those parameters are:
  - The size of the object.
  - The position of the object.
  - The material of the object; determining its colour and translucency.
- Allow users to assign values to the parameters of the environment. The parameters include the following:
  - Position of the camera.
  - Size of the output image and the window frame.
  - Filename of the output image.
  - Colour of the background.
  - Light's colour and position.
- Render multiple objects in the same scene, with at most one light source.

3.2 Optional Requirements

If time allowed, the group established additional requirements. The requirements specified in this category will extend the basic functionality of the ray tracer, thus distinguishing our product from others. The optional requirements are as follows:

- Allow users to add multiple lights to the scene in different positions, colours and intensities.
- Allow users to create other objects such as polygons.
- Allow users to choose different materials, such as plastic, mirror, etc. for the objects.
- Improve the colours of the output image, by using different techniques such as sampling.

- Improve the performance of our system, by applying the multit-hreading concept, if the systems takes a long time while rendering the scene.

Due to time constraints we were unable to implement the use of polygons. However we successfully achieved the other optional requirements.

3.3 Methodology

The methodology the group decided to adopt was the Agile software methodology.

3.4 Architecture

The project task as given to us involved developing the ray tracing engine and the GUI as distinct components which necessitated the creation of a third component to ferry input from the GUI to the Ray tracing Engine. As we were using the ASP.NET technology, it was extremely necessary for the group to take advantage of the ASP.NET MVC web application framework. The different components of the ASP.NET MVC framework are designed in such a way that they can be easily replaced or customised independently without effect on other components (Microsoft). This framework implements the Model–View–Controller pattern and allowed us to divide the application into a composition of three main logical components: Model, View and Controller which represents the Business logic, UI logic and Input logic respectively.

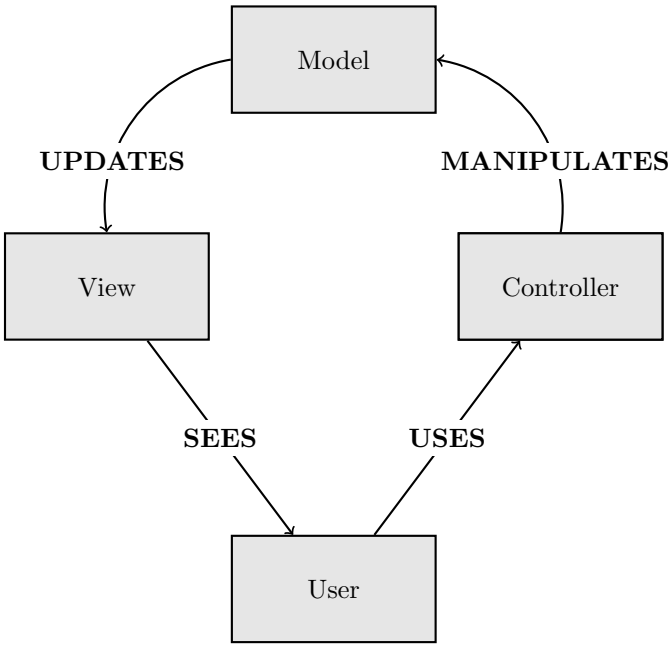

Figure 3.1: Model View Controller Architecture

- **Model** - The Model component corresponds to the back-end and contains all of the business logic. This component is responsible for manipulating all data transferred between the View and Controller components as well as the rules and logic associated with the system. In our system, this is the ray tracing engine that performs all of the calculations and determines if a ray hits an object as well as its colour, shadow, etc [**?**].The data used in this component is gotten from the controller.

- **View** - The View component is used for all the UI logic of the application. This represents the interface that the user interacts with. In our case, this is represented by the front-end which includes an interactive SVG (where users can drag and drop objects as well as move them around), a 3D view which gives actual representation of the scene (including depth) and a form which is used to include additional details such as scene size, light position/intensity, point-of-view, etc. [**?**]

- **Controller** - The Controller acts as an interface between Model and View components. It receives input from the View converts it to understandable commands for the Model. In our system, the controller is represented by an API that extracts data objects from a JSON file and transmits them to the back-end. The controller is extremely useful as it also validates the input from the user. [**?**]

3.5 Prototypes

We produced prototypes to visualisation the design of our system and to evaluate its strengths. Its also assisted us in identify specifications for our system.

3.5.1 First Iteration

Our first iteration is a low fidelity prototype drawn by hand to represent what the homepage might look like, this was our first impression of how it might look.

3.5.2 Second Iteration

Our second iteration was a medium fidelity prototype which include our evolved requirements for the project (i.e. the SVG drag and drop).

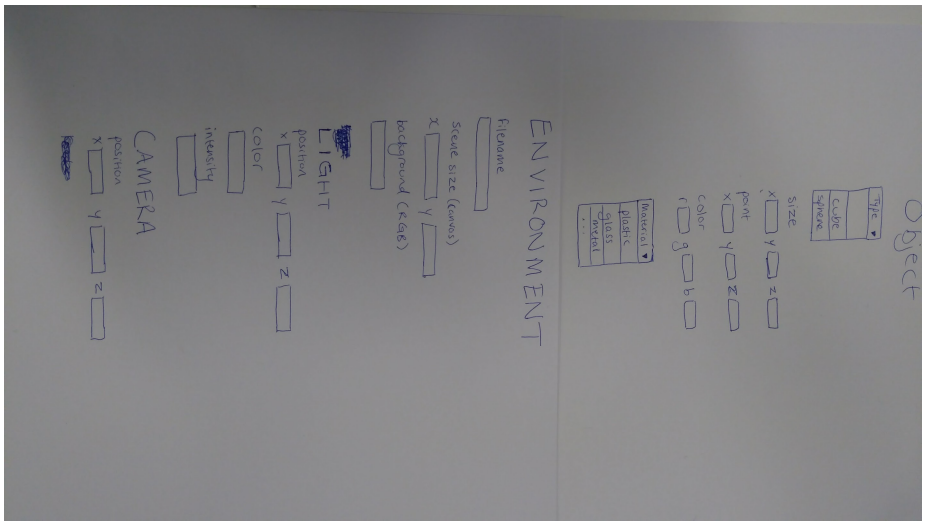

Figure 3.2: First Iteration

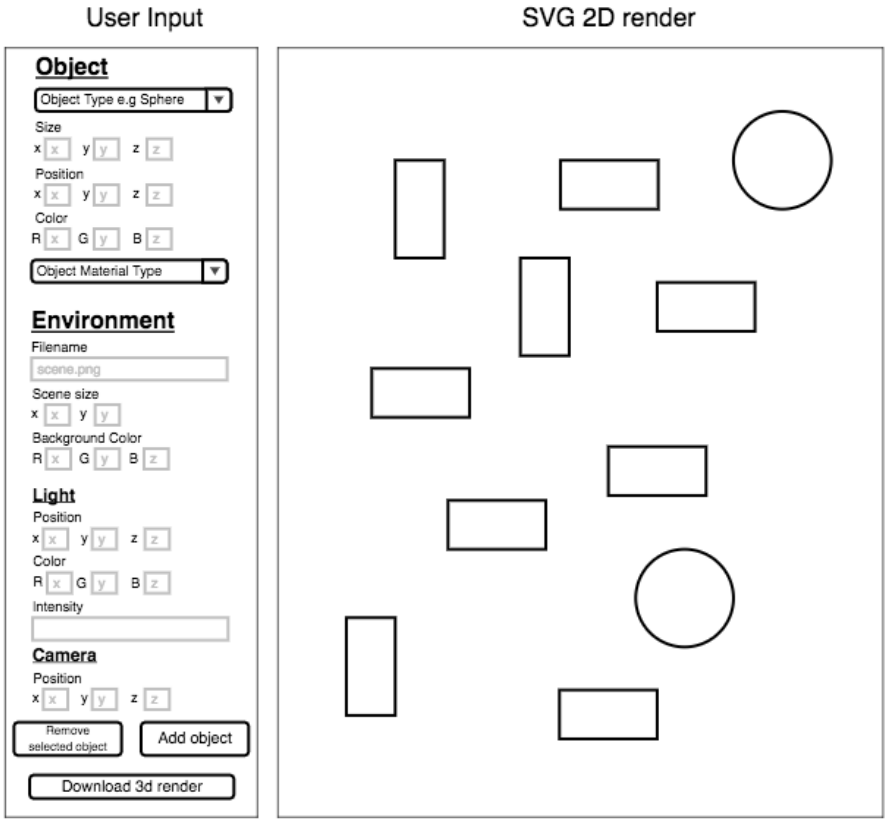

Figure 3.3: Second Iteration

Chapter 4

Implementation

4.1 Back-end

Most of the code in the back-end is based on the book discussed in **Section 2.3.1**. As mentioned before, the ray tracing algorithm is mainly about checking if the ray, which goes from the camera or reflects from objects because of lights, hits/intersects with the object or not. Furthermore, ray tracing depends on the material of each object, and so we should take that into our consideration. In addition to that, each scene has lights and cameras, and since each of these can have different type, we need to implement that to be scalable and open for extension.

In our implementation we make use of different object oriented principles such as abstract classes, inheritance, overriding and Polymorphism. In the following sections, we will discuss in details how we used each of these concepts in our codes.

4.1.1 Models

Our architecture is mainly based on the MVC model as previously discussed in **Section 3.4**. This section will introduce and discuss the models used in our system.

4.1.1.1 Basic Elements

There are many basic elements which are required to build a ray tracer. These elements were reflected in our project as classes:

- **Point3D**: This class is used to create a point in the 3D with x,y and z double values. In this class all the addition, subtraction, division and multiplication were overridden in order to perform them correctly in the context of points. Also, there are other methods such as the distance, which will be used to return the distance between two points.
- **Vector3D**: This class is used to create a vector in the 3D with x,y and z double values. In this class all the addition, subtraction, division and multiplication were overridden in order to perform them correctly in the context of vectors. Also, there are other methods such as the length, which will be used to return the length of a vector. Furthermore, the dot and cross products methods were implemented; since they are crucial in our project.
- **ColorRGB**: This class is used to create a colour with R, G, and B double values. In this class all the addition, subtraction, division and multiplica-

tion were overridden in order to perform them correctly in the context of colours.

- **Point2D**: This class is used to create a point in 2D with x and y double values. No other methods were added to this class; since they are not needed in this class.

- **Ray**: This class is used to create a ray, which will have an origin with type of **Point3D**, and a direction with type of **Vector3D**. This ray will be used in most of the calculations of ray tracing. There are no methods directly implemented in this class, but it will make use of other methods implemented in **Point3D** and **Vector3D**.

4.1.1.2 Configurations

In order to make it easier to change any configured value in our program, we created a class called **Config**, where you can find any default values used in our program. By doing that, we make sure that changing the value in one place will change it in all the places where it is used. For example, the default colour, material, vector, point ... etc are defined in that class.

4.1.1.3 Materials

One of the main factors which will affect the final result of ray tracing is the material of the object. All materials will have a colour, in addition to many different coefficients that might be needed for other materials, such as the diffusion, specular, ambient. In each material, we need a method that will calculate the shades caused by lights and other objects.

It is clear that we need to have an abstract class with the previously mentioned attributes, in addition to an abstract method which will be called calculateShade, which will be implemented differently based on the material. In our solution, we covered three types of materials:

- **Phong materials**: which follows the phong theory such as chalk, metal, plastic. The difference between those materials is the values of diffusion, specular coefficients and the light colour influence.

- **Flat material**: which is the simplest material, that will return the colour of that object regardless to any other environment variables and parameters.

- **Mirror material**: this material is the most complicated one, which will do many different calculations in order to calculate the colour of each pixel of that object.

Each of those materials can have a colour chosen by the user, or just use the default colour, if null. **Figure 4.1** shows the class diagram of the supported materials in our program.

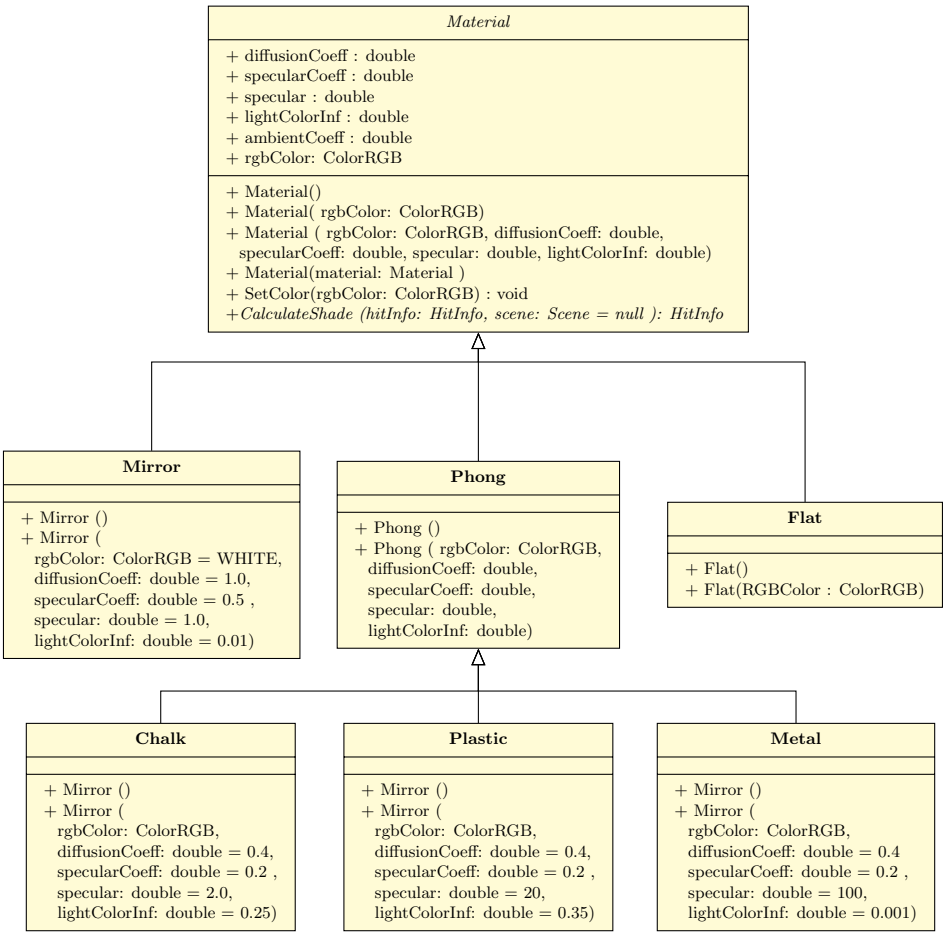

Figure 4.1: Materials Class Diagram

4.1.1.4 Geometry Objects

From the basic idea of ray tracing, we need a method to test whether a ray intersects with the object or not, and since we need our program to be scalable and open for extension, we created an abstract class called geometry object, which will have an attribute called the material; because each object will have a material. In addition to an abstract method called intersect.

In order to add a new geometry object, we need to add a new class, which inherits the GeometryObject abstract class. Then add the attributes needed for the new class, and finally implement the intersect abstract method. The intersect returns an object of type **HitInfo** which will include the following attributes:

- **hasHit**: boolean attribute which be **true** if the ray hits the object, and **false** if not.
- **hitPoint**: it will be the point of where the ray hit the object. This value

will be **ignored**, if **hasHit** is **false**.

- **normalAtHit**: it will be the vector which is perpendicular to the hit-Point. This value will be **ignored**, if **hasHit** is **false**.
- **Ray**: it will be the ray which hits the object. This value will be **ignored**, if **hasHit** is **false**.
- **hitObject**: it will be the object which was hit by the ray. This value will be **ignored**, if **hasHit** is **false**.
- **tMin**: it will be smallest value, which is greater than some epsilon value, from many different possible values, that will be produced when a ray hits an object. It will be used to calculate the **hitPoint**, and it will be **ignored**, if **hasHit** is **false**.

In our program, we have three different objects as illustrated in **Figure 4.2**. Each of these objects has its own attributes, in addition to one of the supported materials, which were listed in **Section 4.1.1.3**.

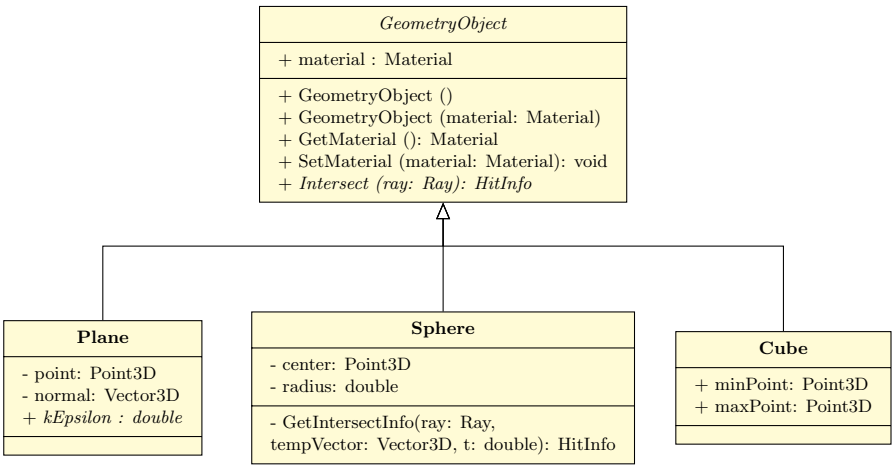

Figure 4.2: Geometry Objects Class Diagram

4.1.1.5 Lights

Each light will have a colour, intensity and a position. There might be different types of lights, and that's why we created a class called **Light**, and it is not an abstract class; since we want to create instances of it. Therefore, we defined it is method to be virtual, and so they can be overridden if we need to change their default implementation. The methods in this class are:

- **GetDistance**: this method will calculate the distance between some point, and the position of the light.
- **GetDirection**: this method will calculate the direction of the light with respect to some given point.
- **GetColor**: this method will calculate the final colour after considering the light effects, such as the colour and intensity.

4.1.1.6 Tracer

The tracer class will be responsible for tracing an input ray with all objects in the scene. Then, if the ray hits one of the objects, it will call the method another time recursively for some predefined depth value. Then, it will use the hitInfo in order to call the **CalculateShade** method, which was previously discussed in the materials section, and the **TraceShadeRay** method, which will calculate the effects of lights on the object.

Finally, the result will be the multiplication of the values returned by the previous two methods in case the ray hits one of the objects, otherwise, it will be the colour of the scene's background.

4.1.1.7 Camera

Another important element of ray tracing is the camera, which represents the way of looking to the objects in the scene. There are many different types of cameras that can be used such as perspective and orthographic cameras. In our implementation, we used the perspective camera. But in order to keep our program open for any future extension, we created an abstract class called camera, which contains:

- **Attributes**:
  - **Position**: which is a 3D point represents the position of the camera in the scene.
  - **lookAt**: which is a 3D vector represents the direction of how to look at the scene.
- **Methods and Operations**: These methods will be implemented differently based on the type of the camera
  - **Render()**: this method will be doing the ray tracing algorithm, by going through all pixels in the window frame and get the colour of that pixel.
  - **FindRayDirection(Point2D point)** : this method will calculate the ray direction from a given point in the window frame with respect to the camera's values.

In the perspective camera's implementation, the **Render** loops through all pixels in the image plane, and then calculate the point, which in order will be used as an input to **FindRayDirection** method, and then call the **TraceRay**, which was discussed before in the previous section. To make the resulting colour smoother, we used one of the sampling techniques called the regular sampling technique.

4.1.1.8 Sampling

One of the enhancements that we did to our output is to use one of the sampling techniques. In our application, we used the **Regular Sampling Technique**. The main idea behind it is to do the tracing on the same square of the image plane, but at different sampling values, and calculate the colour at different

points inside that square for **N** [1] times.

Finally, we sum up these values together, and divide the result by **N** which will be the final colour of that square. By applying this technique, the final colour will be smoother than not using any sampling technique at all.

4.1.1.9 Scene

The scene will be the container of all the previously mentioned classes. Each scene will have a list of geometry objects, list of lights, camera, background colour, sampler, file name which will be the final name of the output file.

The scene will also have many different methods:

- **GetHitInfo**: This method will find if the ray hits any of the objects in the scene, and will return the hitInfo of the nearest object which was hit by the ray, in addition to other values of hitInfo, which were discussed before in **Section 4.1.1.4**.
- **Render**: it will call the render method of the camera's object. More details about this method can be found in **Section 4.1.1.7**.
- **DisplayPixel**: it will take the x and y indices of a point, with the calculated colour as inputs, and then assign that colour to the appropriate index in the final picture pixels result.
- **FinalPicture**: this method will just print the colours in the final pixels array, which are filled by the **DisplayPixel**, that is called by the **Render** method, into an image of jpg type, with the filename received from the front end.

4.1.2 Controllers

In our project, we have one controller called **RenderController**, that will act as a post API. It will receive a JSON from the front-end side, the responsibility of that controller is to parse the received JSON, in a way that can be understood by the back-end's implementation. That's why we are having JSON folder in our project.

The parsing process is done by a method called **ProcessJSON** in **JSONObject** class. We need this class as an interpreter between the front and back ends; since the way front-end dealing with the environment and object values is not the same for back-end. Therefore, for simplicity and to avoid doing a lot of processing calculations and computations on the client side, it is done on the back-end side.

Now and after parsing the JSON to the appropriate format, we can create the scene object, then call the **Render** method. This method will take some time until it renders the whole scene. After that **FinalPicture** method is called, and finally return the image to front-end as JPG file.

[1] The number of samples

4.1.3 Testing the backend

4.2 Front End

One of the main concerns of the team was to create a product that is easy and intuitive to use. In order to enable users to define the different elements they wish to have rendered on the scene, a simple web interface was developed. The interface allows users to enter the different values for the scene elements such as objects with different materials, positions, sizes and colours. Furthermore, the interface allows the user to define the specifications of the scene, even allowing them to add multiple lights at different positions with different colours and intensities.

The web interface has two main modes. One mode shows a 2D representation of the scene and is comprised of two views; a top down and a side view. The 2D view is important as the user is able to view from every angle how the object is visible but it does not give an actual representation of how the object will be positioned once the ray tracing has been produced in the back-end. Hence the other option is the 3D view in which the user is able to change the scene elements including the lights.

4.2.1 Set up

In order to make sure we followed good engineering practices and allowed our code to be easily extendable we implemented the entire client in ES6 styled JavaScript. ES6 is quickly becoming an industry standard with the rise of *NodeJS* and *TypeScript* and thanks to it's modularity, forces us into a neat pattern around which we can structure our code. What is more, because ES6 JavaScript can be ran both locally and in browser, we were able to implement extensive browsing using the *Chai.js* testing framework and incorporate granular unit tests into our continuous integration pipeline - more about this in 4.2.2.

Thought adoption is ever growing, ES6 JavaScript is not supported by a number of current browsers and therefore we chose to transpile and bundle our code using *Browserify* into browser runnable JavaScript. To avoid us having to continuously re-package the bundled code manually we utilised a hot module replacement tool called *Watchify*. *Watchify* monitored each of our included dependencies and whenever changes were saved to a file, the bundle would immediately be rebuilt, ready for us to see our code in action. The combination of an ES6 codebase, hot module replacement, bundling and good unit test coverage made developing the project a frictionless affair.

4.2.2 Testing the client

Because it's all modules it's easy to test - easy to separate logic from front end rendering code We decided to omit the DOM related code from our testing due to the relative simplicity of the code and the brevity of the project.

4.2.3 SVG

The first couple iterations of the project from the client side focused on creating a simple user interface that allows the user to input the most basic objects

and parameters for the scene. By taking this approach we'd guarantee to be delivering a functional product from early in the project life cycle. As we began development, it was immediately obvious that we had to decide on standards that can be used to represent the scene and the objects within it in a concise yet expressive way. During our first meeting we created and documented a JSON structure that would be used to express the render request and sent from the front end to the back end.

The very first task was to first implement the kind of prototype you see in Figure 3.2. We then iterated and largely stuck to the plan

Implemented SVG Wrote functions to map JSON -¿ SVG Focus on being extendible

Implemented drag and drop Talk about how there's libraries out there to do it but we did it ourselfs. Also cite http://www.petercollingridge.co.uk/interactive-svg-components/draggable-svg-element

Tooltip

4.2.4 3D

As mentioned before, the 3D aspect gives an actual representation of the back-end. This was developed using a JavaScript library called Three(three.js). Three is a combination of SVG and WebGL with a simpler way to create objects and all the other necessities such as clicking and labelling. This API is well known as diverse companies[reference] such as Toyota, Porsche or even Google[reference] use it.

Three is quite similar to our back-end as majority of the functions that the back-end provides can be simulated in Three. Functions like adding different materials; adding lights with different intensities and colours; and obviously having a 3D aspect. Difference is, this library allows the user to rotate the scene which can then help them decide how they want the ray tracing image to look like. For example, one user might like an angle better than the other or another user will like the camera position to be higher or lower than the normal view. This gives the user more freedom as opposed to having those two parameters set as default in the background or just having a 2D view which would not be an actual representation.

Chapter 5

Team work

5.1 Process

Weekly meetings At the start of each meeting standup style sync up: Everyone speaks, says what they did, what still needs to be done, calls out any blockers that we can go over in the follow on meeting. Discussion: Talk about what the next steps are, what could be improved, what has blocked us in the previous sprint. Try arrive at outcomes - outcomes made into tickets Next steps: Illicit next steps both from discussion and from product roadmap

5.2 Communication

Communication is one of the most important factors when it comes to working in a software engineering team. For our project we decided to utilise a number of different structured and unstructured communication channels. There exists a large array of great project management and issue tracking tools such as JIRA, Trac or Trello, however, we decided that given the brevity of our project and the relative low number of backlog items (especially at the start), that it would be best to keep our ticket system as close to the codebase as possible. As a result of this, we decided to make use of the inbuilt GitHub issue tracking system. This system does not provide the same calibre of project management tools as some of the forementioned products, however, it does enough for our purposes. We decided to use the GitHub milestone system as a way of scheduling issues for our sprints and implemented a branch naming convention such that we would refer to the issue we're addressing in the name of the branch - {issue number}/{name of branch}. This simple mechanism has worked tremendously because it ensures that on one hand all branches are addressing exactly one tasks or feature and on the other hand, that all tasks and features have been recorded in an issue.

Another way we structured our communication was by the use of pull requests whenever pushing code to master. By raising a pull request and having another team member review it we not only improved our code quality but also spread knowledge of the code base and cross-polinated ideas for features or more efficient solutions. Standout examples of a pull request flow can be seen here: TODO: Cite some nice PR from github.

Not all communication can be done over issue summaries and pull requests and therefore we scheduled weekly meetings to go over what has been achieved, what needs to be discussed and what is the plan for the following sprint. One could say that we condensed our daily scrum meeting, sprint planning, backlog refinement and sprint review all into one weekly meeting. This is not the textbook way of running an Agile project but given that we are not working on the project full time and that these meetings were kept as brief as possible, it still

aligns to the core principles of Agile. During the meetings we would discuss the general trajectory of the project and what we want to accomplish in the next seven days, before breaking off into our sub teams and deciding on the exact issues we needed to create for the week ahead. This way of running a project has both advantages and disadvantages which is covered in section TODO: Cite section that covers this. We mostly met in person but on occasion, when we needed to, we used video conferencing tools such as Appear.In and Gruveo to connect with eachother.

During the course of the sprints, our communication did not fade. We built on some ideas from Extreme Programming and met frequently in our work units to engage in pair programming while working on issues during the sprint. To facilitate the smooth operation of the team we used the lowest friction medium of communication for all of us which was a WhatApp group where we have discussed everything from meeting times, status of different issues or even specific lines of code in pull request.

5.3 Tools

Things that enabled us Visual Studio, Visual Studio Code, Git, Things that kept us on track Travis, Mocha, Istanbull, Things that connected us Appear.In, Gruveo, Whatsapp, Facebook,

Chapter 6

Evaluation

Considering the mandatory requirements that were previously set out, all of them have been accomplished. On the other hand, our optional requirements have been completed as well apart from allowing the users to create extra objects like polygon but instead we have implemented a plane object. However, if the deadline was not a factor, more objects would have been completed as most of them were quite similar to the mandatory ones.

As mentioned earlier, the team was divided into two. This was a success as each team member decided whether they wanted to work in either the front-end or the back-end depending on their own skills,desires or work experience. This was crucial as everyone choose the area they were more comfortable working with which allowed less issues to arise in the future. Each sub-team also had at least one member who had work experience in the particular field. David had experience in the front-end, whereas Othman had experience in the back-end. This was important as both of them were able to lead their sub-team and any problems would be solved efficiently. It also gave the rest of the team members a look into the future as both of their experience helped the team visualise and learn how different code practices and tools are used in the workplace.

Focusing on the mandatory requirements first seemed logical as they were the functionality required for the software to work. Once those were accomplished, the optional requirements would be prioritised. This process went well as we did manage to finish the mandatory requirements. Most of the mandatory tasks were divided in the sub-teams which lead to both individual work and working in pairs. Critical tasks such as SVG implementation on the front-end or creating different shapes in the back-end, was either developed through pair programming or each team member had a small task to do in them, for example one team member would create a cube and the other a sphere. This helped us participate in every aspect of the project and improved our skills such as team work and communication. This also lead to coding problems being solved quickly as two minds are quicker and more knowledgeable than one. On the other hand, small tasks that were also important such as fixing a bug or creating forms, were done individually as not much time was needed for them.

Communication was a major downside that could have been improved between the two sub-teams. This only came to our attention a few days before the first presentation while we were trying to integrate both front-end and the back-end together. It was a small issue at first which became big due to the fact that: variable names were not matching e.g spelling colour in the British way or American way; or the variables that were supposed to be hard coded either in the back-end or front-end e.g the back-end team assumed that the user would input where the camera would be pointing at. While the front-end thought that the back-end team would have it set as default and the user would not be able to

change that as it made sense to always have the camera aiming at the middle of the walls with the user being able to change the position of the camera. This led to some confusion and a lot of changes in the code which wouldn't have been an issue if communication was more effective since the beginning. Communication became better once these issues came out and were pointed out to us but there was still some lack of effective communication.

Throughout each sprint planning, each team member had a task assigned to them which had to be completed before the next sprint. At times, these tasks were not completed due to time constraints or unable to fully complete it cause of some issues they found while developing.This lead to the deadlines being pushed back as another team member had to pick up the task or help out. This was good because everyone had enough time to complete the tasks if there was no issues. It was also bad because at times the work load was not managed well. This caused the internal deadline for the complete development to be pushed back for a week. Fortunately, we had set the internal deadline a month before the project was due which gave us four weeks to write the report. Even though it was pushed, we still had plenty of time to write the report and go over the project one last time before submitting.

Bibliography

[1] An overview of the ray-tracing rendering technique. https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview. Accessed: 2018-03-07.