Light Field Rendering for Real-Time Applications

Sam Cordingley

May 9, 2017

Dissertation Type: enterprise

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Engineering in the Faculty of Engineering.

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of MEng in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Sam Cordingley, May 9, 2017

Contents

1 Executive Summary

In this project, a fully working light field renderer was developed which can be used in any game or film developed in Unreal Engine 4. A system for capturing light fields of computer generated images was also developed.

- I developed a new algorithm for rendering light fields in real-time with a single focus distance for depth of field effects
- I developed a new algorithm for rendering light fields in real-time with depth based focus
- I developed a script for creating a light field camera within Autodesk Maya
- I developed a script for compiling images rendered in Autodesk Maya so that they can be used by the custom renderer within Unreal Engine 4.
- I researched and designed additional features for the light field renderer

The project was very successful, resulting in a system that allows high quality path traced graphics to be viewed in real-time within the context of a game. This is something that previously could not be done.

2 Supporting Technologies

- Autodesk Maya 2016 was used to model and render scenes
- Pixar Renderman For Maya was used to render scenes
- The maya.cmds Python wrapper for MEL was used to create scripts to run in Maya
- The Pillow Python library was used to create image manipulation scripts
- C++/Blueprint used for CPU code in Unreal Engine 4
- HLSL was used for the shader code in Unreal Engine 4

3 Acknowledgements

Thank you to my supervisor, Dr. Carl Henrik Ek, for providing advice and input on the design of the algorithms and writing of the dissertation.

4 Contextual Background

In the early days of real-time rendering, the scope of graphics was severely limited by the hardware of the time. Those developing real-time rendering engines for applications such as games, were more concerned with the problem of rendering whatever was possible in order to fulfill the design of the gameplay. Developed by Id Software in 1993, Doom was one of the first games to take advantage of real-time 3D rendering [12].

Figure 1: Doom (1993) in-game image

Although the rendering in this game was advanced enough to allow for the innovative three dimensional gameplay envisioned by the designers, many of the graphics within the game were still just two dimensional and the game was very far off from anything that could be considered an imitation of reality. The idea of rendering physically correct, realistic looking 3D graphics in real-time was something that was out of the question for many years.

Meanwhile in other applications such as animated films, rendering appeared to be much more advanced, with films such as Pixar's Toy Story being released just a couple of years after the original Doom game.

There was a stark difference between the two applications of 3D rendering. Although the stylised look of Toy Story could not be considered to resemble reality, the detail of the characters and the realistic looking lighting effects made for much more convincing visuals.

Figure 2: Toy Story (1995) rendered image showing advanced lighting techniques including reflection

The difference in rendering between these two applications was down to render time. In games, the rendering must be done in real-time so each frame needs to be rendered quickly, limiting the amount of computation that can be done. In a theoretical sense, graphics are very simple. The physics of how light travels is known and so it should be easy to produce realistic images. The problem arises from trying to produce images with limited time. In animated films, this is not so much of a problem as rendering is not done in real-time.

The main practical difference is that in animated films, the visuals are largely limited by the artists, whereas in games of the time, the results of the rendering could not reach anywhere near the vision of an artist, so were dictated more by the rendering techniques and the capability of the hardware.

In recent years, this disparity is still present, though much smaller due to significant advances in hardware and rendering techniques. A number of techniques have been developed to attempt to diminish the limited time problem. One such technique is light mapping, first implemented in the game Quake. Light mapping involves storing lighting data in a texture, which is then combined with the original textures. The advantage of this is that lighting does not have to be computed in real-time, however it cannot be used for dynamic lights [9].

This technique became more popular as the amount of memory in GPUs

increased. More memory allows for more data to be precomputed and stored in a way in which it can be accessed quickly. These kind of techniques not only allow for more realistic lighting in games, but also give the artists more freedom. However, despite these advances, the disparity between the real-time rendering used in games and the offline rendering used in films still exists, albeit much smaller.

Outside of games and film, a type of image technology called light fields has been explored in recent years. In contrast to regular images that describe the light arriving from different angles at a single point, a light field describes light arriving from different angles through an area. This means that a light field has two extra spatial dimensions, as well as the two angular dimensions used to describe a regular image [13].

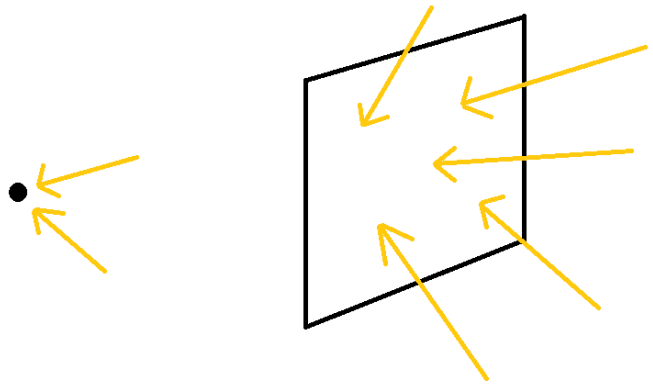

Figure 3: Left: Light arriving at a single point (regular image), Right: Light arriving at an area (Light field)

The concept has seen commercial success in the form of light field cameras.

Figure 4: Lytro (Commercial Light Field Camera)

These cameras take advantage of a micro-lens array to capture an image from many positions, while also encoding the direction of the ray of light that corresponds to each pixel [15]. The aim of this kind of setup is to capture all of the light rays in a scene coming from every direction. This allows you to

manipulate the final image in a number of ways that would not be possible with a conventional camera. First of all, it allows you to refocus the image at different distances by only choosing the pixels which correspond to rays originating from the chosen focus plane. It also allows you to reconstruct the image from a range of positions by only choosing the pixels which correspond to rays which would eventually arrive at the chosen view point. This concept is powerful as it gives photographers the freedom to change the photo after they have already taken it [14].

The idea of this project was to take this concept and explore how it could be applied to the discussed rendering disparity problem. This was done by applying the idea of light field technology to computer generated scenes. This allows scenes to be rendered beforehand and then viewed from different positions in real-time in a similar way to how a light field camera works. This enables high quality, Pixar film level rendering to be integrated seamlessly within a real-time application such as a game, essentially closing the gap between real-time and offline graphics.

As well as bringing higher quality graphics to games, this technology could also allow for more interactivity within animated films. Because a traditional animated film is rendered with a single viewpoint for any particular point in the film, it means that the viewer has little freedom in terms of what they want to look at, it is all decided by the creators of the film. However, with the rising popularity of virtual reality, there is now much more demand for more freedom for the viewer. Ideally, a viewer would want to be placed within the film and be able to look in any directions and even move around. This is not possible with the traditional way of making films, however it is with games. There have been attempts to make films more suitable for virtual reality, the main one being '360' video. With this technique, the film is captured with a three hundred and sixty degree field of view and then projected onto the inside of a sphere. The problem with this is that it only allows for freedom in rotation. The viewer can only view the video from a single point.

If the film was instead captured as a light field, the viewer would have freedom not only in rotation, but also in translation. With the help of virtual reality, this would allow the viewer to move their head or even walk around within the film, all with correct lighting and parallax effects. This viewer freedom would dramatically increase immersion and make for a much more realistic experience.

To achieve all of this, a system must be developed to capture computer generated light field images and view them within a real-time engine or more specifically, a game engine. This would allow light fields to be integrated into a game and also allow for the development of interactive films.

More specifically, the main objectives of this project are:

1. Capture Light field images of scenes in the animation software: Autodesk Maya
2. Develop a fully featured light field viewer in the real-time graphics engine: Unreal Engine 4
3. Investigate compression of light fields to reduce file size
4. Add further features to the light field viewer that could be useful in a real-time application.

While a huge amount of work is being done regarding Light Fields, a system like this designed for use within real time games is not currently available and could be very useful for game developers looking for a way to improve the look of their games, without sacrificing performance.

5 Technical Background

In the physical world, everything we see is a result of light originating from a source, propagating throughout a physical space and finally entering our eyes. Light travels in a straight line until it encounters a surface, then the behaviour of the light is dictated by the physical properties of the surface. In general, light can either reflect off a surface, refract through a surface, or be absorbed by the surface, or in the case of light sources, be emitted. Opaque surfaces can be put into three main categories: mirror, glossy, and diffuse. Mirror surfaces only reflect specularly. This means that when light is reflected off a mirror, or purely specular, surface, the angle of the reflected ray is identical to the angle of the incident ray [6].

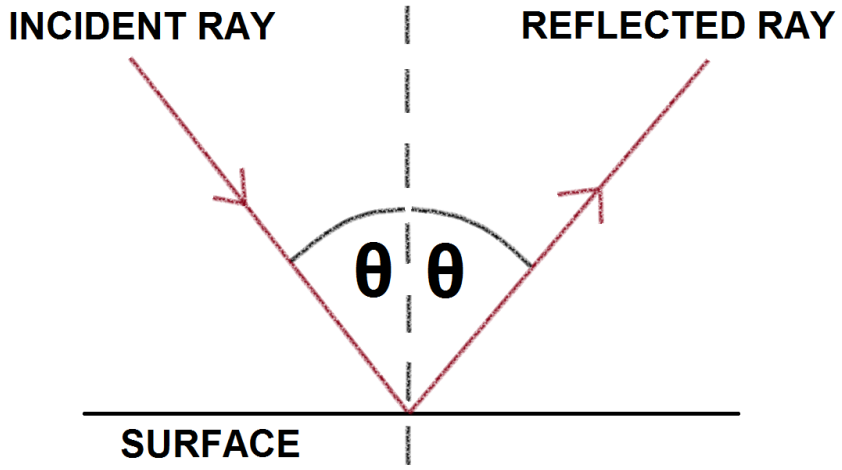

Figure 5: Single ray of light reflected off a mirror surface

In the case of diffuse surfaces, the angle of the reflected ray is independent of the incident ray. Light is reflected in many different directions, this is usually due to the light first bouncing around within the internal structure of the material before exiting it [6]. However, surfaces are not only purely diffuse or purely specular, many exhibit both properties as seen in figure 7. The reflective properties of a material can be defined by a bidirectional reflectance distribution function (BRDF). The BRDF describes how much of the incoming light reflects at a certain direction [7].

The glossy category of material is an example of a material that exhibits both of these properties. A glossy surface may have random perturbations and imperfections that cause the rays of light to be reflected in different directions,

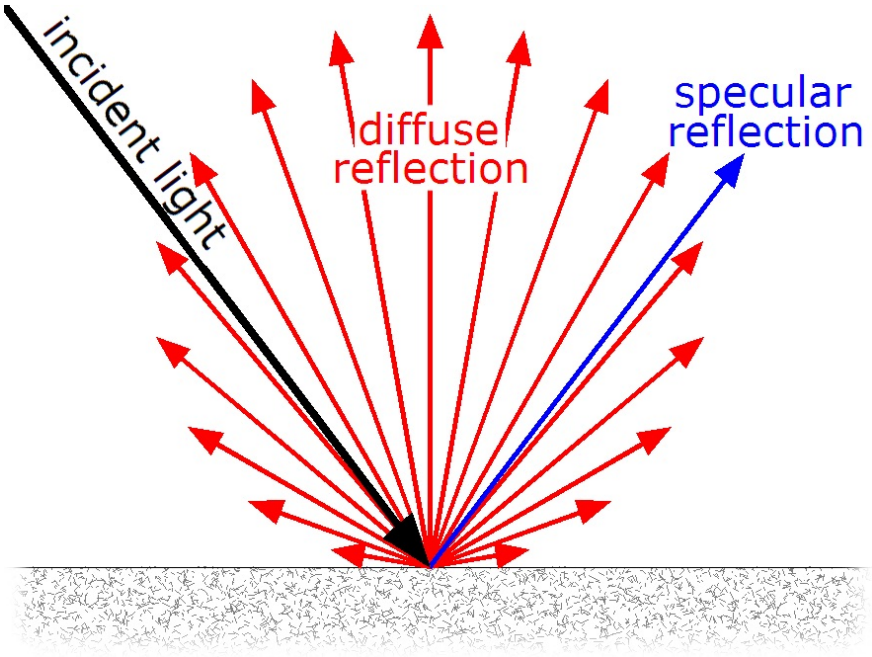

Figure 6: Diffuse reflection (license: https://commons.wikimedia.org/wiki/File:Lambert2.gif)

but the average direction of reflected rays will still resemble specular reflection.

When light is reflected, it's colour can be influenced by the material, leading to colour bleeding, which causes the colour of nearby surfaces to be influenced by one another [5].

It is the fact that light can reflect in many directions that makes simulating difficult. The number of rays that have to be tracked increases exponentially with the number of bounces as after a bounce, a single incident ray can contribute to many reflected rays.

When simulating illumination, what we want to know is the radiance leaving any point in the scene in any direction. In this context, radiance is the total power and colour of the light. Due to the law of conservation of energy, it is known that the total light energy entering a point is equal to the energy absorbed plus the energy exiting the point.

It is due to this law that the properties of illumination can be formalised mathematically. An equation formalising the problem of finding the radiance leaving any point and any direction was introduced by James Kajiya in 1986 in the form of "the rendering equation" [11]. The rendering equation is an integral equation that states that the radiance at a certain point and direction is equal to the emitted radiance at that point plus the sum of incoming radiance from all directions multiplied by the BRDF and a weakening factor. The weakening factor is to account for the fact that when light encounters a surface at an angle,

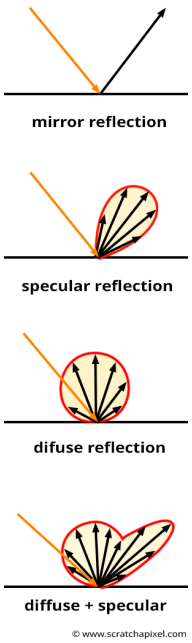

Figure 7: Reflection behaviour of different materials

it is smeared across a larger surface area than when it encounters the surface perpendicularly.

The equation can be written as:

$$L_{\mathrm{o}}(\mathbf{x},\, \omega_{\mathrm{o}},\, \lambda,\, t) \;=\; L_e(\mathbf{x},\, \omega_{\mathrm{o}},\, \lambda,\, t) + \int_{\Omega} f_r(\mathbf{x},\, \omega_{\mathrm{i}},\, \omega_{\mathrm{o}},\, \lambda,\, t)\, L_{\mathrm{i}}(\mathbf{x},\, \omega_{\mathrm{i}},\, \lambda,\, t)\, (\omega_{\mathrm{i}} \cdot \mathbf{n})\; \mathrm{d}\, \omega_{\mathrm{i}}$$

Where

- L is radiance
- $\mathbf{x}$ is the position in the scene
- ω_o is the direction of the outgoing light
- ω_i is the direction of the incoming light
- λ is the wavelength of the light
- t is time
- n is the normal of the surface (normalised vector orthognal to the surface)
- Ω is a hemisphere centered around $\mathbf{x}$ representing all possible values of ω_i

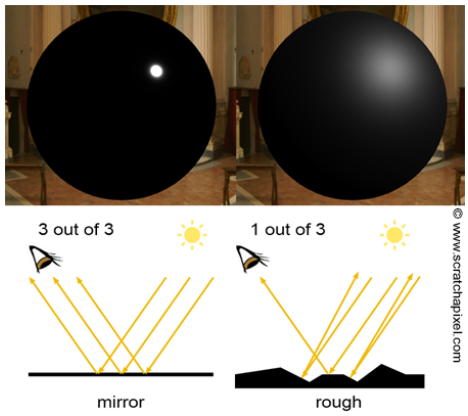

Figure 8: Left: Mirror, Right: Glossy

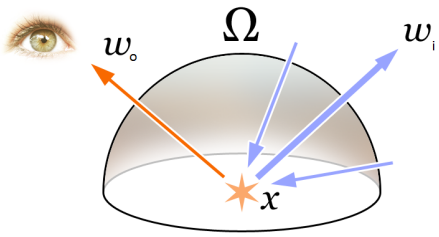

Figure 9: Visualisation of the rendering equation [16]

It is evident that the radiance function L appears on both the left-hand-side and within the integral of the right-hand-side of the rendering equation. This means that the rendering equation is a Fredholm integral of the second kind [2] and therefore cannot be solved analytically.

Rendering techniques that attempt to solve this equation are known as global illumination techniques because they simulate indirect diffuse lighting, illuminated all parts of a scene, not just those in direct line of sight of a light source.

The best tractable algorithm for numerically approximating the rendering equation is called path tracing and is also described in Kajiya's rendering equation paper. It is a variation of ray tracing.

5.1 Ray tracing

Ray tracing renders scenes by tracing the path of individual rays of light using the known discussed properties of light [1]. Tracing individual rays of light means that advanced effects like refraction, reflection and scattering can be achieved easily. The problem with this method is that compared to other rendering techniques like rasterisation, it is extremely computationally expensive

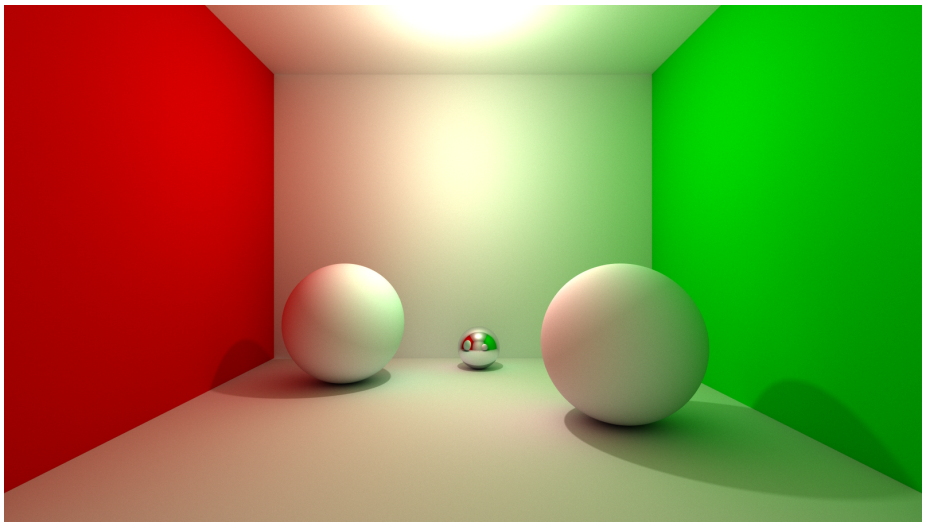

Figure 10: Example of an image rendered with global illumination effects

and in practice can take a long time to render. This makes ray tracing generally unsuitable for real-time rendering. However, it is used heavily in offline rendering for animated films.

The simplest ray tracing algorithm would cast a ray through each pixel, and find the intersections of each ray with the mathematically defined geometry of the scene. This is usually done by describing the geometry as many triangles. The algorithm would test the intersection of each ray with each triangle by treating each triangle and a plane and each ray as a line intersecting that plane. This allows the intersection to be calculated using simple linear algebra. If the position of the intersection is within the triangle then an intersection is valid. For each valid intersection the radiance at that point can then be calculated by casting rays to each light source in the scene. If a ray is not blocked between a point on geometry and a light source, then that light source contributes to the radiance of that point. If it is blocked, it means that the point at which the ray originates is not within view of the light side and therefore in the shadow of that light source.

The rays of light are traced backwards in this way because we only care about the rays that go through each pixel. If the rays were traced the other way round, the majority of the rays would not be received by the viewer anyway.

With this simple algorithm alone, a scene can be rendered, however for the advanced effects mentioned, a slightly more complex version of the algorithm must be used. Instead of only calculating the first intersection of a ray, a recursive version of the algorithm is used to follow multiple bounces of the light rays.

However, neither of these variations can solve the rendering equation as they do not simulate the global illumination effects of indirect diffuse lighting, all of

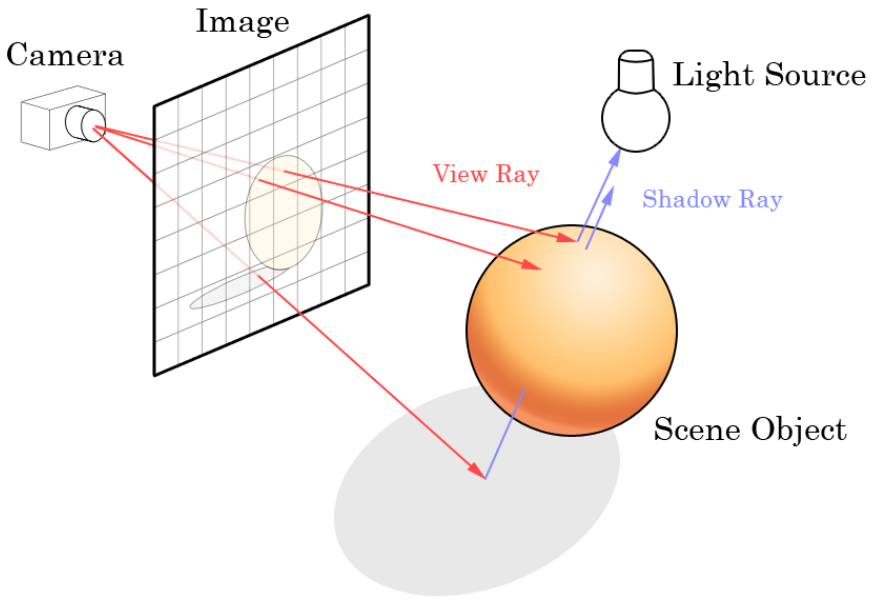

Figure 11: Ray tracing diagram. [16]

the lighting at each point in the scene comes directly from the defined light sources.

To achieve this, a rendering algorithm would have to trace the reflected rays in all directions. However, this would be extremely computationally expensive as the number of rays to track would exponentially increase as discussed. Although render time is not as important for offline applications, it is still a concern. For example when producing an animated film, the studio still would like the rendering to be as fast as possible so that their film can be completed on time and within budget.

5.2 Path tracing

Path tracing is a Monte Carlo method which allows the global illumination computation to be reduced while still remaining realistic and physically correct [11]. It is essentially a numerical approximation to the solution of the rendering equation. The reflected rays are sampled using the BRDF as a probability distribution function. This means that realistic images can still be generated even with a small sample of rays.

As well as tracing rays originating from the view point, rays are also traced originating from the light sources, this is so that the algorithm can converge more quickly. Path tracing is considered the most realistic feasible rendering technique and is used in Pixar Renderman, which is used in this project.

5.3 Light Field

Path tracing is usually used to capture all of the rays of light entering the single point of a camera. However, the same technique can also be used to capture all of the rays of light entering an area, or light field.

In the original light field rendering paper, written by Mark Levoy and Pat Hanrahan in 1996 [13], the light field is defined as 'the radiance at a point in a given direction'. This definition is equivalent to that of the 5D Plenoptic function, describing a ray of light by it's position in three dimensions, as well as it's direction, parameterised by two angles.

However there is redundancy in this function when dealing with regions of space with no occluders. In this case, as a ray of light is not blocked, it's radiance is constant along it's line. This key observation means that one of the dimensions of the plenoptic function is completely redundant, allowing it to be reduced to a 4D function, which is the light field.

This reduction is what allows light fields to be efficiently rendered as it saves on both data storage and complexity of reconstruction. The disadvantage of this reduction is that it restricts the views of a scene to outside of it's convex hull.

5.3.1 Representation

This 4D light field function can be parameterised in a number of ways. The most common parameterisation, and the one described in the paper, is to use two planes, between which a line can be described by connecting a point on one plane to a point on the other. These two planes are known as the uv plane and the st plane and usually correspond to the camera plane and the focal plane respectively.

In practical terms, the camera plane is the plane on which the original rays are sampled from, this is the plane on which the cameras used to capture the image lie. The focal plane is the plane on which reconstructed rays are focused.

This representation is known as a light slab. Multiple light slabs can be combined to provide a greater range of views.

5.3.2 Reconstruction

From this representation, a view of a scene can be constructed by finding the rays of light that most closely correspond to the rays of light that should arrive at the view point. This is done by projecting a ray onto a focus plane and then finding the ray in the closest cameras that also go through that point on the focusing plane. The pixel value is interpolated between the pixels found from each of the closest cameras.

To capture the images of a light field, similar rendering techniques of that used in high end animated films can be used. A lot of these films now use ray tracing or path tracing to render scenes. In this project, images were rendered using Pixar's Renderman renderer within Autodesk Maya.

5.4 Real-time rendering

On the other hand, the rendering techniques of the real-time world are much less computationally expensive, but more "messy". As they cannot afford to simulate the physics of light transport, they must rely on other methods to make realistic looking images.

Rasterisation is an example of a real-time rendering technique. Instead of simulating light rays in three-dimensional space, all major calculations are done in two-dimensional image space. This is achieved by projecting the three-dimensional geometry onto the view plane.

The geometry of the scene is modeled by triangles. This is because it ensures that all faces of the geometry are planar. This is different to a quadrilateral for example, where not all of the vertices of the geometry necessarily lie on a plane. If a face is not planar, it is not clear how it should be visualised in the renderer.

The vertices of each triangle are projected into 2D screen space by modeling the view camera as a pinhole camera with perspective projection. Lines are then drawn between the vertices to form the triangles. The lines are drawn in screen space, an algorithm such as Bresenham's line drawing can be used to do this [8].

First the position on the image of each point of geometry is found using perspective projection. Then lines are drawn between the points to form the triangles of the scene geometry.

5.4.1 Vertex Shader

Illumination can be calculated by finding the distance between each point and a light source. In this step other per-vertex calculations can be made, such as a world space offset.

5.4.2 Pixel Shader

The radiance can then be interpolated first along the lines, and then between the lines to calculated the illumination of the whole triangle. Local coordinates of the points can also be interpolated along these lines, and then interpolated between the lines to find the local coordinates of each pixel in the triangle. These are called UV coordinates and are used to map textures onto each triangle. The colour values found from the texture mapping and the colour values found from the illumination can then be blended together to get the final colour value.

Rasterisation is the main technique used in real-time game engines, such as Unreal Engine 4, which is used in this project.

5.5 Unreal Engine 4 and DirectX

Unreal Engine 4 is a real-time game engine developed by Epic Games [4]. It is written in C++ and uses Microsoft's DirectX graphics API. A graphics API provides a game engine with direct access to the GPU where the rasterization calculations are performed. Unreal Engine allows shaders to be written in the

HLSL language for DirectX which can be run on the graphics card. A shader is a program that defines how a surface in the geometry is shaded. In the simple rasterization example, the shader calculated the distance to a light source and interpolated the radiance across the surface.

The two main types of shaders are vertex shaders and pixel shaders. Vertex shaders calculate the properties of a vertex and interpolates between vertices to calculate the properties of each pixel, while pixel shaders calculate the properties of a pixel directly.

Both types of shader are used simultaneously as they are both suitable for different uses. Code written in a vertex shader is generally much faster as it only has to be calculated for each vertex, but this also means that advances effects such as bump mapping and shadows cannot be achieved. Pixel shaders are slower and they have to be calculated per pixel but provide a lot more freedom for advanced effects.

5.6 Autodesk Maya

Maya is 3D computer graphics software developed by Autodesk [3]. It allows for the creation and rendering of 3D models and animations. Maya comes with two embedded scripting languages, it's own, MEL (Maya Embedded Language), and Python. Both have the same use and are designed for customising the functionality of the software.

It can render models and animations created in the software with path tracing using the NVIDIA Mental Ray renderer, or Pixar Renderman, available as a plugin.

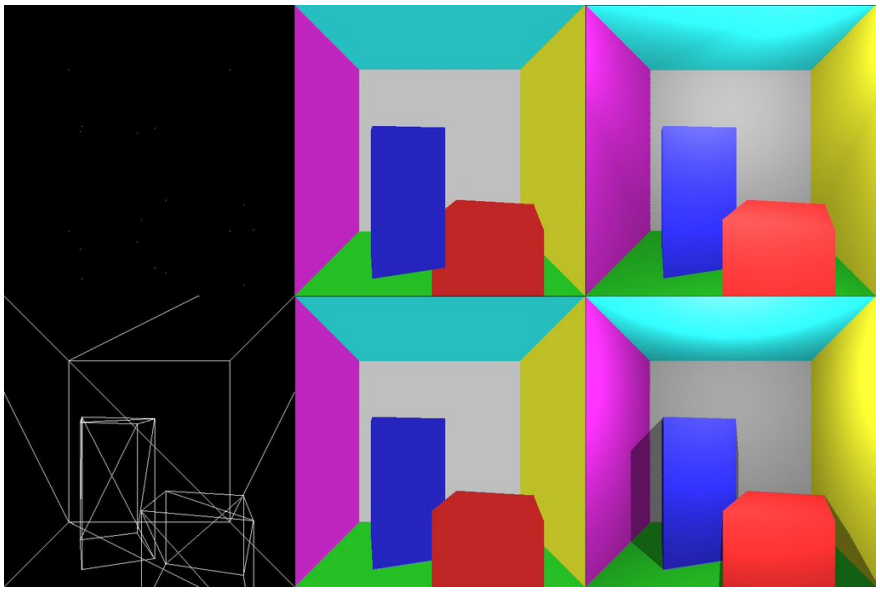

Figure 12: The steps of the rasterisation rendering process

6 Project Design

The process of designing the system begun with extensive research into the topics of light fields and real-time rendering.

After research, the first step was to design the overall component pipeline of the system. The basic pipeline had to consist of

1. A 3D Modeling program
2. An offline renderer to render the 3D models
3. Software to build a light field using the renderer
4. A real-time renderer to render the light field within

The specific software packages and technology components used to fulfill this design went through a few iterations.

1. Autodesk Maya (A 3D Modeling program)
2. Pixar Renderman (An offline renderer to render the 3D models)
3. Python with the Pillow library (Software to build a light field using the renderer)
4. Unreal Engine 4 (A real-time renderer to render the light field within)

Figure 13: The pipeline with specific software choices

Autodesk Maya was chosen because it is the most popular 3D modelling software within the animation industry. It works with a vast number of different renderers, including Pixar's RenderMan. Furthermore, the highly customisable nature of Maya would allow it's functionality to be extended to enable the capture of light fields required for the system.

The other overall design problem within this initial step in the pipeline was to figure out how to use Maya and RenderMan to produce the images needed for light field. The final design involves using the built in Maya scripting functionality to produce a virtual light field camera within Maya itself. This method was chosen because it allows users of the system to visualise the light field camera and easily set it's parameters as though it was a regular Maya camera. It also allows the user to render using any renderer of their choosing using the built in Maya rendering functionality.

After the images have been rendered, they need to be combined into a single light field image. While this could have potentially been done as a plug-in for Maya, the aim was for the system to be as modular as possible. Writing this part of the system as a separate program would allow it to be used with other rendering programs and also allow it to be used without having to open Maya.

This part of the system was developed in Python and utilised the Pillow library to import, manipulate, and export images.

After the light field is compiled into a single image, it can easily be imported into Unreal Engine 4 as a texture.

Unreal Engine 4 was chosen because it is used extensively both by game development companies and independent developers. It is also free to use and it's source code is available to view and edit allowing for extension of functionality. It can also run on a wide range of hardware, including mobile devices.

I had never used Unreal Engine 4 before so this meant I had to spend time learning a new game engine from scratch.

7 Project Implementation

For this project, a real-time light field renderer was implemented. The renderer allows planar light fields to be reconstructed and displayed in real-time within Unreal Engine 4. Two versions of the renderer were implemented. One uses a focus plane to choose the light field rays. This allows for a depth of field effect but also means that the whole scene cannot be in focus at once. The other version uses the depth, or distance from the camera, of each pixel to focus rays on the actual objects in the scene, rather than a focus plane. This means that the whole scene can be in focus. Both versions have their advantages and disadvantages and are useful for different applications.

As well as the renderer, python scripts for generating and compressing the light fields were implemented.

On top of this, extra features were either implemented or explored to provide greater functionality for the light field renderer within the context of a real-time application.

7.1 Light Field Camera

The virtual light field camera was developed as a Python script within Maya. The script allows the user to convert an existing regular camera into a light field camera.

7.2 Light Field Image Compilation

7.3 Light Field Rendering

The light field renderer runs on the GPU and was written as a pixel shader within Unreal Engine 4. It consists of a "Light Field Plane" that can be placed in the real-time scene.

7.3.1 Focus Plane Version

1. Project a ray through each pixel of the Light Field Plane
2. Find the intersection of the ray with a focus plane some distance behind the Light Field Plane
3. Calculate the position of each camera of the light field
4. For each camera:
   (a) Calculate it's position
   (b) Project a new ray from the found intersection to the camera position
   (c) Find the intersection of this ray with the image plane of the camera
   (d) Find the pixel at the intersection

5. Weight pixels based inversely on the distance of the camera to the original intersection on the "Light Field Plane"

The goal of the light field rendering algorithm is to find the rays of light that most closely correspond to the rays of light that should arrive at the viewer at it's current position. For example, if the viewer is positioned exactly at the location corresponding to one of the cameras used to generate the light field on the UV plane, and it's focal length was exactly the same as the focal length of this camera, all of the rays that should arrive at the viewer also arrive at this single camera.

However, as the viewer moves back from the UV plane, more of the captured scene should come into view and not all of the rays of light required arrive at a single camera. Instead, rays that arrive at multiple cameras of the light field have to be used.

In a practical sense, each ray of light is encoded as a pixel in the light field image. The light field image is made up of many smaller images, each corresponding to one of the cameras on the UV plane. The position at which a light ray arrives is given by the position of the camera that captured it's corresponding pixel. The direction of the light ray corresponds to the position of it's corresponding pixel in the image. For example, if a pixel is exactly in the centre of an image, it's direction is exactly perpendicular to the UV plane. If a pixel is away from the centre, it's arriving at the UV plane at an angle. The precise angle of the ray can be calculated with knowledge of the focal length and the dimensions of the camera's film gate. If the dimensions of the film gate are known, then the position of the pixel on the film gate can be easily found by dividing it's position in pixels by the resolution of the image and then multiplying by the dimensions of the film gate. With this value, and the value of the focal length of the camera, we have the opposite and adjacent lengths of a right angled triangle and can therefore simply calculate the angle using the inverse sine of the ratio of these two lengths.

$$\theta = \sin^{-1} \frac{d}{FocalLength}$$

$$d = \frac{FilmDimensions * PixelPosition}{Resolution}$$

For a given viewer position, one could calculate the position and direction of the ray of light for each individual pixel of the light field image. Any pixels that correspond to rays of the light that would arrive at the viewer, based on their position and direction, could then be used for the viewer image.

However, the number of correct rays will most likely be quite low as we are only considering the rays that should arrive at the exact position. In practice, this strategy would only work if there are a very large number of very high resolution cameras. For example, if we had a camera on the UV plane for each pixel of the viewer image, and each had a very high resolution, there would be enough rays arriving at a single point in order to form a complete image.

This is not practical as it requires a huge amount of data to be stored. Therefore, an alternative strategy with only a limited number of cameras was devised.

7.4 Focus Plane

Instead of finding perfect rays from the light field image, we can instead look at all of the rays that we need to make up the viewer image, and then find rays in the light field that most closely correspond to each of the required rays. Doing it this way means we can build a complete image even with a small number of cameras.

Similar rays are found with the use of a focus plane.

The focus plane is a plane parallel to the UV plane but a distance away from it. This distance will be referred to as the focus distance. First, the rays that we need are found by casting a ray out from each of the pixels in the view image. For each ray, the intersection with the focus plane is found. Then, a ray between each camera and the intersection point is found. For each ray, the intersection point is found with the image plane of the corresponding camera, from which a pixel can be found.

In general, line-plane intersections can be found either algebraically or parametrically using matrices. The first iteration of the algorithm calculated the intersections algebraically as the required matrix operations were not available in HLSL in order to calculate it efficiently on the GPU, for example, there is no inverse function.

$$p = l_0 + l\frac{(p_0 - l_0) \cdot n}{l \cdot n}$$

Where l_0 is a point on the line, l is the direction of the line, p_0 is a point on the plane, and n is the normal of the plane. The normal is a vector orthogonal to the plane with length of 1.

However, in later iterations of the algorithm a much more efficient method of calculating the intersections was found. The intersection calculation could be simplified because of the fact that the focus plane and the image planes are parallel to the UV plane. This meant that once the initial rays between the viewer position and the UV plane were found, the other intersections could be easily found by simply multiplying them by ratios. For example, the point of intersection on the focus plane for a ray could be found by multiplying the ray between the viewer and the UV plane by the ratio of the shortest distance between the viewer and the focus plane and the focus distance.

However, a dot product method may still have to be used for different parameterisations such as a spherical light field.

After a pixel from each camera has been found, the pixels can be weighted based on the distance of their corresponding camera to the UV plane intersection. Cameras that are closer to the intersection receive a higher weighting as their pixels are more correct. The area around the intersection in which cameras are considered, acts as a kind of virtual aperture. With a large aperture, depth

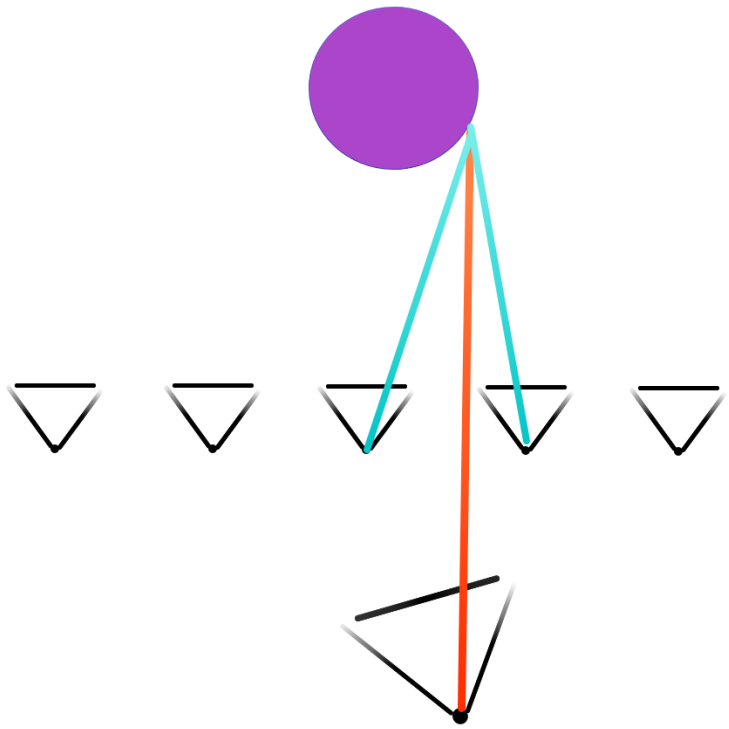

Figure 14: Similar rays

of field effects can be achieved, similar to the behaviour of a regular camera. It can also help to smooth the transitions between cameras.

In a perfect, continuous, light field, the final pixel is calculated using this aperture equation:

$$P = \frac{\int_C p(\mathbf{x},\, \omega,\, f,\, c_i)\,(\alpha - \sqrt{(\mathbf{x}-c)^2})^s \,\mathrm{d}\, c_\mathrm{i}}{\int_C (\alpha - \sqrt{(\mathbf{x}-c)^2})^s \,\mathrm{d}\, c_\mathrm{i}}$$

Where

- $\mathbf{C}$ is the UV plane, where the cameras lie
- $\mathbf{c}$ is the position of a camera on the UV plane
- p is a function that chooses the pixel in camera c_i given the position of the intersection on the UV plane $\mathbf{x}$, the angle of intersection ω, and the individual camera position on the UV plane c

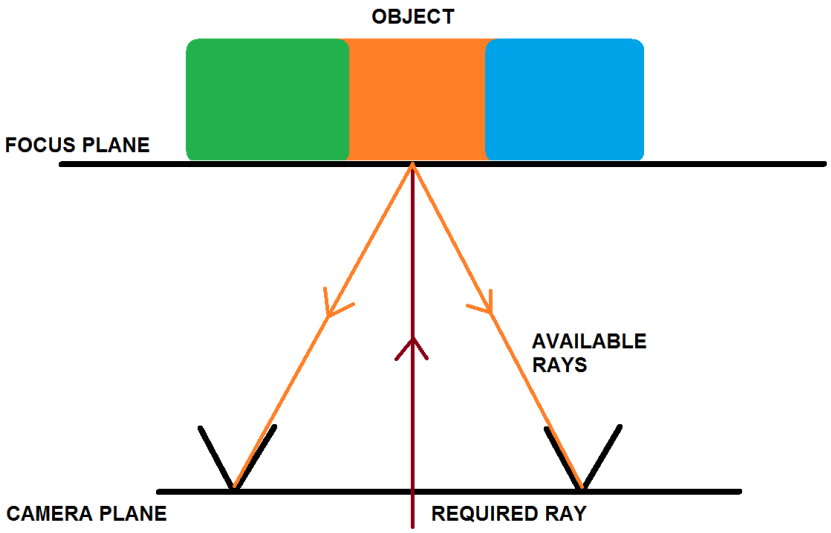

Figure 15: Focus Plane

- α is the radius of the aperture
- s is the aperture exponent, higher values create a smoother aperture w, and the focus distance f

7.5 Depth of Field

Depth of field is the effect achieved when only objects in the scene within a small range of depth are in focus, in this case, only pixels that correspond to parts of the scene that are at the focus plane are correct. It is often a desirable feature of a renderer but is hard to achieve in traditional real-time renderers. In this light field renderer, the depth of field effect comes naturally as a consequence of using a focus plane and virtual aperture without significantly impacting performance. This is because if the part of the scene is not at the focus plane, rays that are chosen to reconstruct the required ray end up corresponding to pixels in different parts of the image, resulting in a blurred mix between the two pixel values. This is demonstrated in figure 17 where the correct value of the pixel would be orange, but because the focus plane is behind the object, the wrong rays are chosen, resulting in a mix of green and blue.

However depth of field may not always be desired, and is hard to avoid using the focus plane algorithm.

In a regular camera, reducing the size of the aperture will diminish this effect. A pinhole camera produces images with everything in focus, for example.

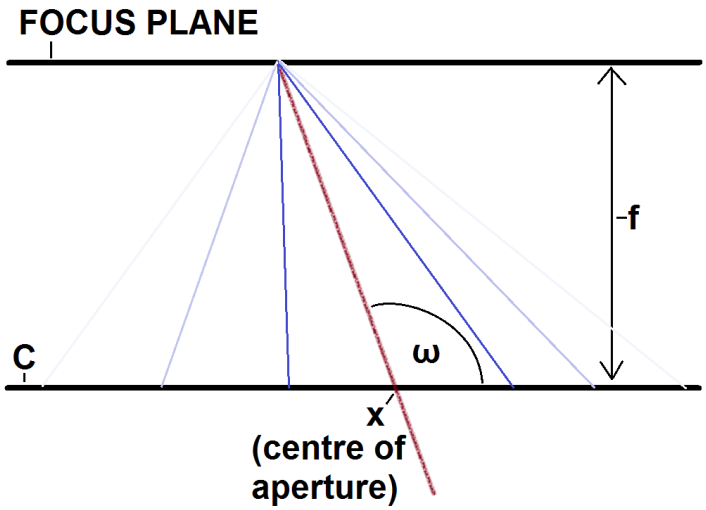

Figure 16: Visualisation of the variables of the aperture equation. The darkness of the purple lines indicate the weighting of that ray, the rays that intersect cameras closer to the intersection point are weighted more highly as they are more similar to the desired (red) ray

However, in the light field renderer, reducing the size of the virtual aperture will diminish the blurring effect, but incorrect pixels will still be found.

One way to improve this is to use multiple focus planes. This would allow multiple parts of the scene to be in focus at once.

7.6 Depth

When generating the light field, capturing depth information as well as colour can have a number of advantages. The main advantage is that it can be used for focusing, allowing all parts of the scene to be in focus at once. This allows rays to be focused on the actual scene, rather than a single, flat focus plane.

The depth based focusing algorithm uses a depth image to search for the best position to focus the light field rays. This is done by starting with an initial focus plane in the centre of the scene. Using this focus plane, a pixel is found in the depth image of the closest camera to the intersection of the camera plane. If this depth value is different to the depth of hte focus plane, it means that the focus plane is not yet in the correct position. More specifically, if the depth value is less than the depth of the focus plane, the focus plane needs to be moved towards the camera. If the depth value is greater than the depth of the focus plane, the focus plane needs to be moved away from the camera.

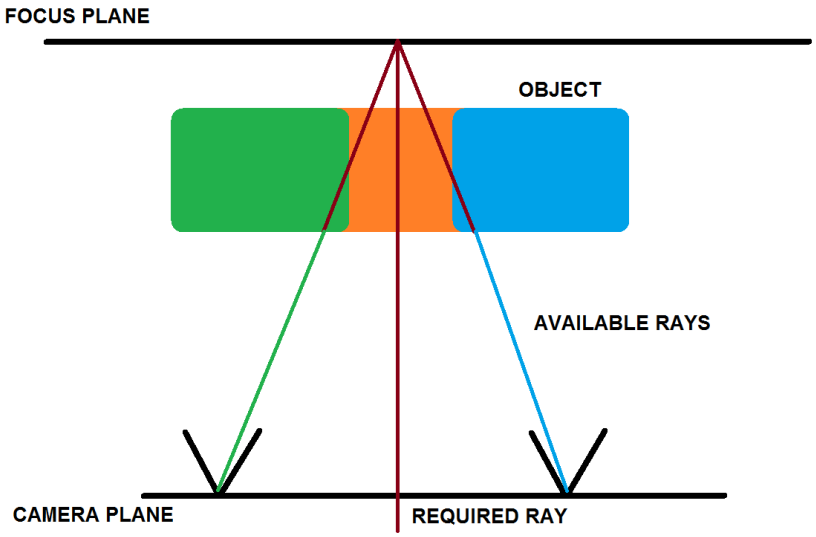

Figure 17: Incorrect Focus Plane

7.6.1 Depth Based Focusing

1. Project a ray through each pixel of the Light Field Plane
2. Calculate the position of the closest camera to the position of the pixel on the light field plane
3. Repeat recursively:
   (a) Find the intersection of the ray with an initial focus plane some distance behind the Light Field Plane
   (b) Project a new ray from the found intersection to the closest camera position
   (c) Find the intersection of this ray with the image plane of the camera
   (d) Find the pixel at the intersection
   (e) Get the depth value of this pixel
   (f) Compare pixel depth value with depth of the focus plane
   (g) If the pixel depth is greater than the depth of the focus plane, move the focus plane halfway between it's current position and the maximum focus distance
   (h) If the pixel depth is less than the depth of the focus plane, move the focus plane halfway between it's current position and the minimum focus distance

4. For each camera:
    (a) Calculate it's position
    (b) Project a new ray from the found intersection to the camera position
    (c) Find the intersection of this ray with the image plane of the camera
    (d) Find the pixel at the intersection
5. Weight pixels based inversely on the distance of the camera to the original intersection on the "Light Field Plane"

This algorithm has a time complexity of O(log(n)) where n is the number of depth values of the image. This means that it scales well with higher precision depth images. The number of iterations can also be lowered to achieve a quicker, approximate result.

7.7 Shader Optimisation

7.7.1 Dynamic Branching

In shaders, conditionals can have a big impact on performance. In a pixel shader, if the conditional takes the same branch for every pixel, it is called static branching and has no extra cost. Dynamic branching is when a conditional takes a different branch for different pixels.

Modern GPUs have a very large number of cores. For example, the AMD R9 390 GPU used to develop this project has 2560 cores. These allow the GPU to compute in parallel using thousands of threads. Furthermore, these threads are grouped in hardware as a "warp" or "wavefront". All of the threads in a wavefront can process a single instruction at the same time. The instruction is exactly the same for every thread, but the input data can differ.

When threads in a wavefront can't all execute the same instruction, 'divergence' occurs. Divergence is a common consequence of dynamic branching. This can have a large impact on performance on some GPUs, such as on mobile, but not so much on modern desktop GPUs.

7.8 Compression

When the light field image is imported into Unreal Engine, it is compressed using DXT5, which is a lossy compression algorithm. DXT5 works by first splitting the image into 4x4 pixel segments. As each pixel is 32 bits, each of these segments is 512 bits before compression. A palette of 4 colours is used to represent the colours of the 16 pixels in a segment. The final pixel values will be assigned by an index to a colour in this palette. This index is only 2 bits because there are only 2^2 different colours. In the palette, only two colours are stored, these are the two extremes. The other two colours of the palette are calculated by interpolating between these two extremes. The colours are stored with compression so that each only uses 16 bits. The original image used 24

bits for each colour, 8 bits for each of the 3 channels. The alpha channel is compressed in a similar way, but uses 8 values per palette. Again for the alpha channel, only the two extremes are stored, each with 8 bits. However because there are 8 values in the palette, a 3 bit index must be used per pixel [10]. With this compression method, each of the 16 pixel segments now only use 128 bits (2*16 bits for the colour palette, 16*2 for the colour indexing, 2*8 bits for the alpha palette, 16*3 bits for the alpha indexing). Considering the original image used 512 bits per 16 bit segment, this is a compression ratio of 4:1. As an example a 219727 Kb light field image is compressed to 54382 Kb.

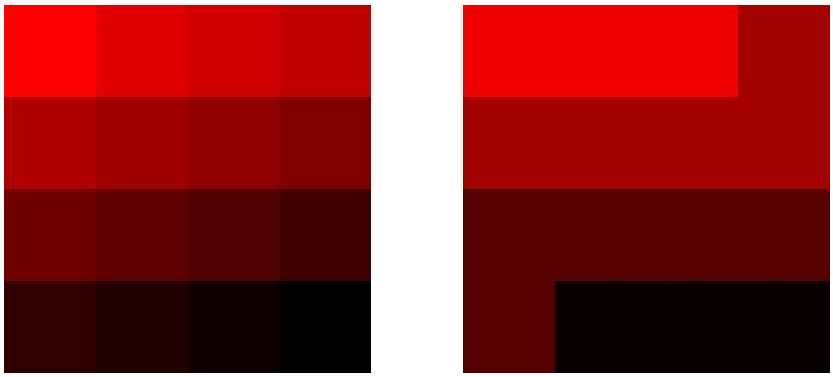

Figure 18: Before and after comparison of DXT compression on a 16 pixel segment [10]

8 Evaluation

8.1 Results

To test the light field renderer, a test scene file for Maya and Renderman had to be found. The scene file found contains a highly geometrically complex model made up of 2476773 polygons. A single frame of this scene was rendered at a resolution of 500x500 using a regular single camera setup. The rendering of the frame took 4 minutes and 34 seconds in total.

This scene takes this long to render for a number of reasons. The first of which is that it is using a path tracer renderer. This allows for realistic global illumination effects but is very slow when compared to real-time renderers.

The other main reason is the complexity of the geometry. The higher the number of polygons in the geometry, the more intersection calculations the path tracer has to compute. Complexity of geometry is also a major issue in real-time renderers.

Clearly, the high render time of this single frame means that this scene could not possibly be rendered in real-time. To render this scene at 30 frames per second, which is considered a minimum for games, the rendering would have

Figure 19: Test scene render

to be completed in 33 milliseconds. The render time is 8303 times slower than it needs to be in order to run in real-time.

However, instead of rendering with a single camera, this scene was then rendered as a 15x15 camera light field. The total render time for the light field was about 17 hours but could then be viewed in real-time from different angles, as though the scene was being rendered in real-time.

Another scene was also rendered. This scene is simpler and took a lot less time to render, but more clearly shows the benefits of a light field over a single camera image.

In figure 23, it can be seen that when viewing the light field from different positions, the image changes and looks correct from both positions. On the left image, the box is not occluded by the ball, but in the right image the box is occluded by the ball due to the angle and position at which it is viewed. It can also be scene that in the right image, more of ball is in light than in the left image. This is because a different section of the ball is being viewed.

When comparing this to a single camera image, texture mapped onto a plane,

Figure 20: Light field viewed from the left

Figure 21: Light field viewed from the centre

Figure 22: Light field viewed from the right

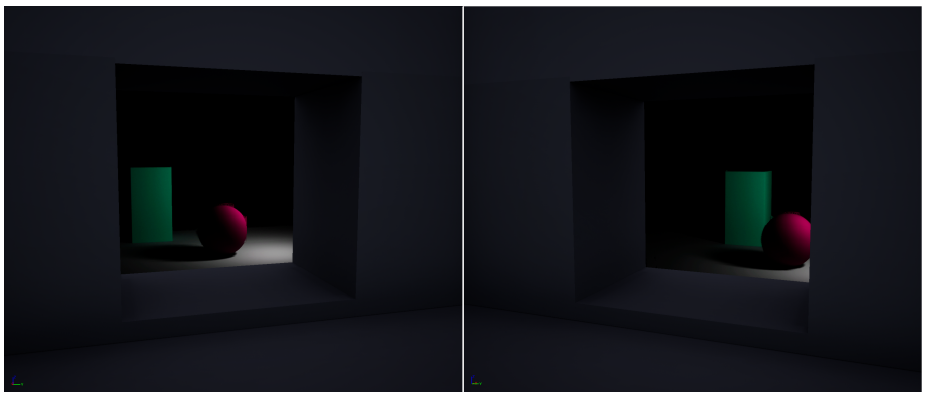

Figure 23: Light field viewed from two different positions

it is evident that neither of these effects are present.

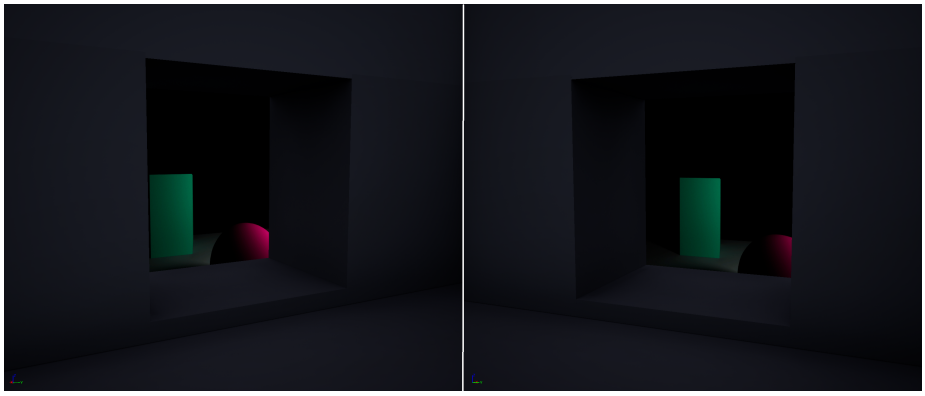

Figure 24: Texture mapped single camera image viewed from two different positions

For a single camera image, there is only one position at which the image looks correct.

Real-time dynamic objects can be placed within the light field scene and the will intersect and be obscured correctly by the prerendered objects of the light field scene. This is because every pixel is offset by the depth of that pixel.

An issue with the light field renderer can also be seen in figure 23. Around the edges of the objects there are some incorrect pixels. This is probably due to the fact that the depth image is only 8 bits, providing 256 levels of depth. In the scene in figure 23, the maximum depth of the image is 10 metres, this means that there is 3.9 centimetres between each depth value. This may be causing parts of the image to be slightly out of focus. This could fixed by rendering the depth image with a higher bit depth.

8.2 Performance

To evaluate the performance of the two light field rendering algorithms, the time to render one frame of a game using these algorithms was measured. As a control, the frame render time of a texture mapped static image was measured, coming in at 6.14 milliseconds. This represents the simplest form of rendering images from which the overhead of the light field rendering can be measured. The render times of the single focus plane renderer and the depth based focus renderer came in at 7.27 milliseconds and 8.38 milliseconds respectively. For comparison, most games are expected to run at a minimum frame render time of 33 milliseconds (30 frames per second), while about 17 milliseconds is considered a good frame render time.

This means that the single focus plane renderer increased render time by only 1.13 milliseconds, while the depth based focus renderer increased render time by only 2.24 milliseconds.

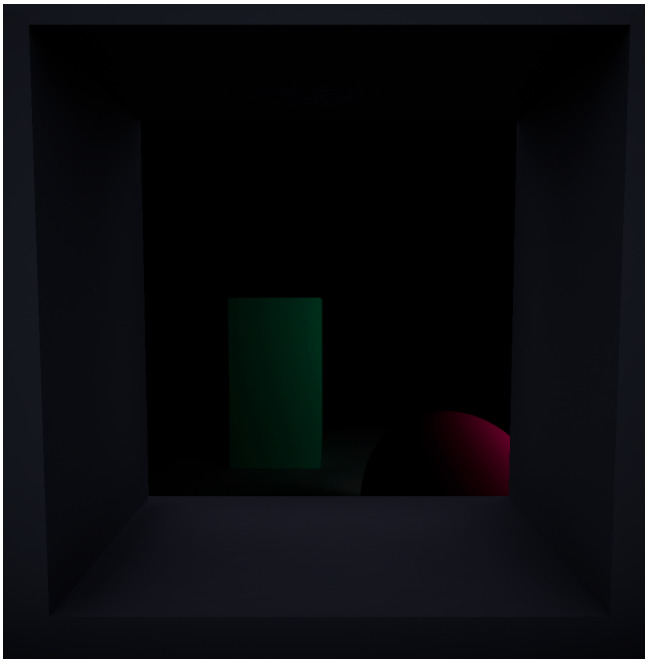

Figure 25: Screen shot of regular texture mapping with a render time of 6.14 ms resulting in 162.95 frames per second

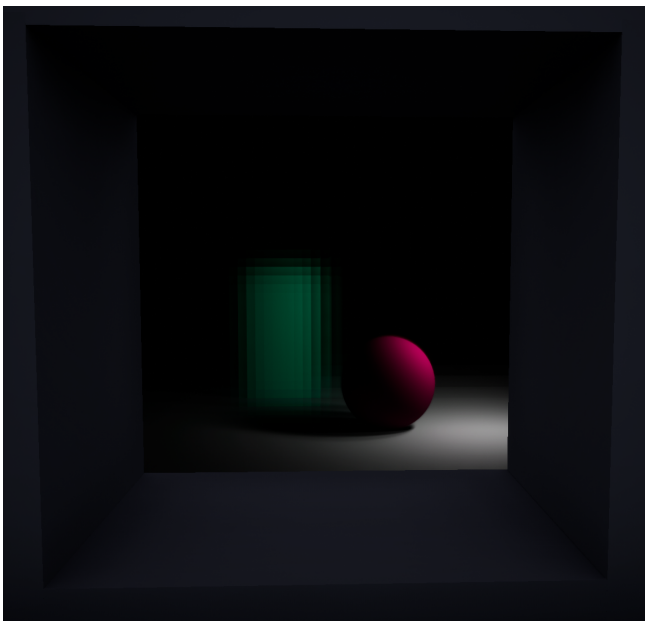

Figure 26: Screen shot of the single focus plane light field rendering with a render time of 7.27 ms resulting in 137.50 frames per second

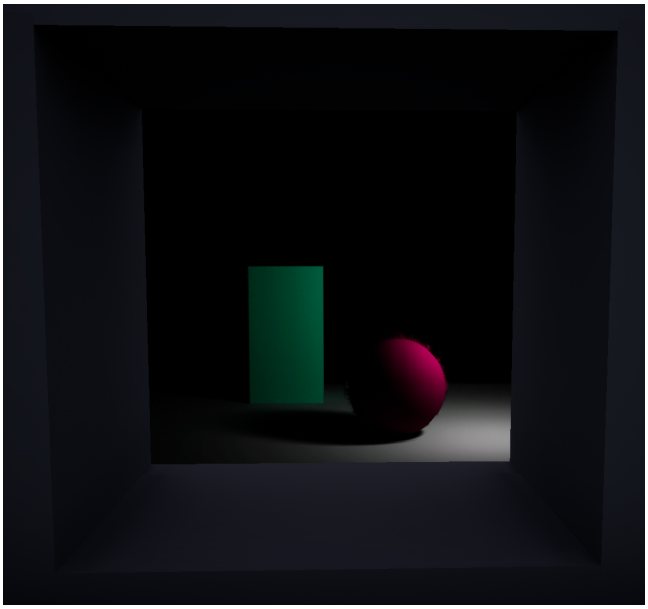

Figure 27: Screen shot of the depth based focus light field rendering with a render time of 8.38 ms resulting in 119.28 frames per second

If this is put into the context of a typical game, running at 30 frames per second, these overheads would contribute to 3.4% and 6.8% of the total render time respectively.

9 Conclusion

The light field rendering system described has a huge amount of utility for many applications. In games it can be used to render parts of a scene with complex geometry and global illumination, something that previously could not be done in real-time. Although there are limitations, such as limited field of view and lack of interactivity, there are still many ways in which it could be implemented seamlessly within a game. For example, it could be used to render a window to an outside scene. As well as the computer generated light field images, the system could be adapted to take advantage of light fields capturing the real world. This could be useful for assets within a game that are particularly difficult to render, even with offline rendering, such as fire.

The system could also be used to create animated films. The whole film could be rendered and captured as a light field which would then allow the viewer to move around within the film as it is playing. This would be especially useful for virtual reality films where the six degrees of freedom that the light field system provides is essential to a realistic viewing experience.

The light field renderer can easily be used in Unreal Engine games in the form of a plugin. While the light field capture system can easily be used to capture other scenes within Maya with the use of the light field camera creation script.

All of the original objectives of the project have been fulfilled.

1. Captured Light field images of scenes in the animation software: Autodesk Maya
2. Developed a fully featured light field viewer in the real-time graphics engine: Unreal Engine 4
3. Investigated compression of light fields to reduce file size
4. Added further features to the light field viewer that could be useful in a real-time application.

9.1 Future Work

9.1.1 Lighting Integration

Although real-time objects can be placed within the light field scene, they do not currently take on the lighting of the light field scene. There are two completely separate lighting environments between the light field scene and real-time scene. Real-time objects could be seamlessly integrated with the use of a light probe technique. Light probes are usually used in conjunction with light mapping

in order to integrate dynamic objects with the light mapped static objects of a scene. This involves placing many light probes throughout a scene. Each light probe measures the radiance in all directions at a single point. These light probes can then be used to interpolate the radiance continuously throughout the scene in order to light dynamic objects.

A similar technique could be used for light fields. During the rendering state in Maya, light probes could be placed throughout the scene. Then, as well as exporting the rendered images, the position and light data of each light field could be exported. This data could then be imported into Unreal Engine and used to calculate the lighting for real-time objects within the light field scene. The lighting for a real-time object would be creating a new probe positioned in the centre of the real-time object. The radiance at this probe is calculated by finding the four surrounding probes so that the new probe is contained within a tetrahedron with a light probe at each corner and then interpolating between the light probes.

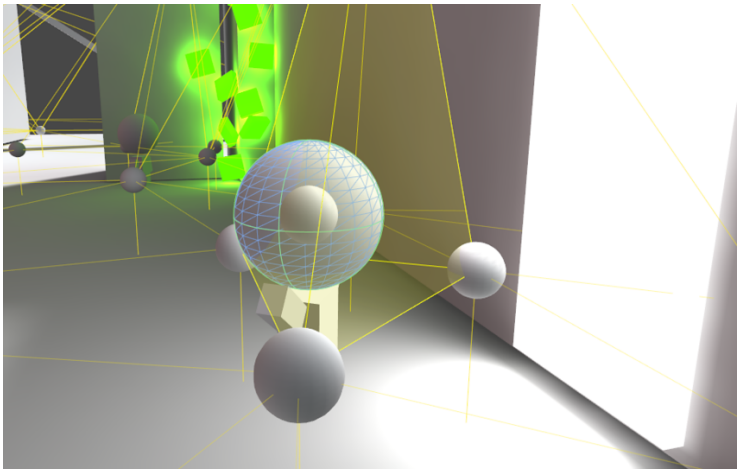

Figure 28: Light probes for light mapping in the Unity game engine. The small spheres represent the light probes, while the larger sphere is the dynamic object to be lit. The shaded yellow area is the tetrahedron from which the new light probe is interpolated

9.1.2 Physics Integration

Again, while real-time objects can be placed within the light field scene, they cannot interact physically with the scene. This is because the actual geometry data of the scene has not been stored. However, the depth at each pixel in the light field image is stored. This could be used to calculate collisions allowing for basic physics interactions between real-time dynamic objects and the light field scene.

9.1.3 Animation

Instead of reproducing just a static scene, this system could be extended to animated scenes extremely easily. It would involve rendering not just one image for each camera of the light field, but one image for each frame in an animation for each camera. The light field renderer could then step through each image sequentially to play an animation. This is the only change that would have to be made.

9.1.4 Spherical Light Fields

The light field renderer could be adapted from a planar parameterisation to a spherical parameterisation. This would allow the renderer to reconstruct light fields formed by cameras on a sphere. The cameras could face inwards towards an object, allowing for the object to be viewed from all directions. They could also face outwards, providing a full view of a scene, allowing the viewer to look in any direction. This would be especially useful for animated films.

The system would have to be adapted in a number of ways. Firstly, a Maya script would have to be written in order to facilitate the creation of a spherical light field camera within Maya. On the Unreal Engine side of the system, the renderer would have to be adapted quite heavily. As mentioned, the planar version allowed for optimisations that would not work for a spherical light field. This would also mean that the spherical light field renderer would be slower.

References

[1] *http://eqworld.ipmnet.ru/en/solutions/ie/ie-toc4.htm.*

[2] *https://www.autodesk.co.uk/products/maya/overview.*

[3] *https://www.unrealengine.com/what-is-unreal-engine-4.*

[4] *http://www.3drender.com/glossary/colorbleeding.htm.*

[5] Scratch a pixel. *Introduction to Shading.*

[6] Arthur Appel. *Some techniques for shading machine renderings of solids.*

[7] Walt Disney Animation Studios Brent Burley. *Physically-Based Shading at Disney.* 2012.

[8] J.E. Bresenham. *Algorithm for computer control of a digital plotter*, volume 4. 1965.

[9] Keshev Channa. *Light Mapping - Theory and Implementation.* 2003.

[10] FSDeveloper.com. *DXT Compression Explained.*

[11] James T. Kajiya. *The rendering equation.* 1986.

[12] Hank Leukart. *The Official Doom FAQ.*

[13] Marc Levoy and Pat Hanrahan. *Light Field Rendering.* 1996.

[14] Lytro. *First Generation Lytro Light Field Camera: Overview.* 2016.

[15] Lytro. *How does the Lytro Light Field Camera work?* 2016.

[16] Timrb. *[GFDL (http://www.gnu.org/copyleft/fdl.html), CC-BY-SA-3.0 (http://creativecommons.org/licenses/by-sa/3.0/), GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 4.0-3.0-2.5-2.0-1.0 (http://creativecommons.org/licenses/by-sa/4.0-3.0-2.5-2.0-1.0)], via Wikimedia Commons.*