

UNIVERSITY COLLEGE LONDON

MSC PROJECT DISSERTATION

Differentiable Ray Tracing for Designing Optical Parts

Candidate Number: QJFF4

September 12th, 2022

Abstract

The goal of this project was to investigate the design of optical parts with differentiable ray tracing and modern machine learning techniques. The optical component is specifically designed as an augmented reality headset made up of pinholes of varying sizes and that is placed 1cm from an image source. The differentiability of the optical component is made possible through the PyTorch library, and the ray tracing aspect is sped up through a graphics processing unit called CUDA. The renderer is created with the help of the ODAK computer graphics and visual perception library.

The algorithm would render a visualisation of looking through the optical component when the pinholes are placed randomly and then use stochastic gradient descent to optimise the component until a clearer image is generated. The result of testing the system on 8 images gave an average difference of 2.8×10^{-3} between the test image and the image that is seen when looking through the optimised aperture array component.

Table of Contents

Abstract	2
Chapter 1: Introduction	5
1.1. Project Motivation	5
1.2. Aims and Objectives	6
Chapter 2: Literature Review	7
2.1. Differentiable Rendering	7
2.2. Inverse Graphics	8
2.3. Near Eye Displays	9
2.3.1. Waveguide	10
2.3.2. Holographic designs	10
2.3.3. Pinholes	11
2.4. Coded Apertures	11
2.5. Optimisation	13
2.5.1. Algorithms to reduce the number of rays	14
2.5.2. Algorithms to reduce the number of objects tested against each ray	14
2.6. Artificial Intelligence and Ray Tracing	15
Chapter 3: Theory	17
3.1. Ray Tracing	17
3.1.1. The intersection of a ray with a plane	19
3.2. Optimisation	21
3.2.1. Constrained Optimisation	21
3.2.2. Unconstrained optimisation	22
3.2.3. Gradient descent	22
Chapter 4: Design	26
4.1. Image-pre-processing	26
4.2. Initialising the light sources	26
4.3. The Aperture Array	26
4.4. The Forward	27
4.5. Selecting the loss function	27
4.6. Optimisation	28
4.7. Python Libraries	28
4.7.1. Graphical processing units (GPUs) and PyTorch	28
4.7.2. ODAK	29
4.7.3. Image processing libraries (OpenCV, Matplotlib, Pillow)	29
Chapter 5: Results and Analysis	30

5.1.	Initial results with a randomised array	30
5.2.	Optimisation Results	32
5.3.	Percentage reduction in MSE loss.....	33
5.4.	Analysis	34
Chapter 6: Conclusion and Further Improvements.....		35
Chapter 7: Bibliography.....		37
Chapter 8: Appendix		43
8.1.	Terminology.....	43
8.2.	Convergence History.....	43
8.3.	Project Timeline.....	45
8.4.	Code.....	45

Table of Figures

Figure 1: Main components of a near eye display [21]	10
Figure 2:How Holographic displays work [20].....	11
Figure 3: A Fresnel Zone Plate [39].....	12
Figure 4: Different types of image processing with neural networks [51].....	15
Figure 5: (a) The Pinhole camera model, (b)Computer Graphics Pinhole model [58]	17
Figure 6: Ray Tracing techniques, forward ray tracing and backwards ray tracing [59]	18
Figure 7: The distance between the intersection points and the surrounding pixels. In this image, the intersection point would be assigned to pixel A, as it has the smallest distance between them.	20
Figure 8: Visualisation of a convex and differentiable function undergoing gradient descent [60].....	23
Figure 9: Flow chart of system design	26
Figure 10: (a) 28 x 28 Aperture Array for grayscale images with values ranging from 0 to 1, (b) 28 x28 x3 array for coloured images.....	27
Table 1: Result of Ray Tracing through randomised aperture array	31
Table 2: Result of Ray Tracing through optimised aperture array	33
Table 3: Percentage reduction in losses	34
Figure 11: Initial project Plan	45

Chapter 1: Introduction

1.1. Project Motivation

Near-eye displays are a massive growing industry projected to reach USD 5.3 billion on a global scale by the year 2027 [1]. The displays also known as head-mounted displays (HMD) allow its users to immerse themselves in a world of virtually rendered images (VR) or virtual images overlapped with the real world (AR). They can, depending on the specifications respond to movement from the users and allow for virtual object interaction and manipulation. Near-eye displays can be applied to a wide and diverse range of problems and have been used to transform industries such as health care, communication, manufacturing, and gaming among others.

Designing near-eye displays come with a multitude of issues such as limited field of view, bulkiness, low optical resolution, and ease of manufacturing [2]. These issues usually come at a trade-off each other, and much research has been done in finding a solution that addresses all the issues. One of the areas of research involves the use of pinhole cameras. The pinhole camera is the earliest known camera model and it consists of a lightproof box with a hole which allows light to pass through it and project an inverted image of the object in front of the box.

When designing near-eye display systems, using ray tracing for modelling before manufacturing offers the advantage of processing pixels independently thus making it easy to integrate with an image-based tracking technique. This means that the rendered images will coincide with where the eye is currently looking [3]. It also means that augmented reality is a great candidate for applying ray tracing as pixels come directly from an RGB camera, meaning that the number of processed pixels is highly limited [4].

Historically, ray tracing had the disadvantage of being computationally expensive and made real-time rendering slow and inefficient. Thus, rendering methods such as rasterization were preferred over it however, recent developments in software and hardware devices allow for the optimisation of ray tracing techniques.

Therefore, the purpose of this project is to develop an AR display system by combining the differential and computational advantage of ray tracing with the simplicity of pinhole

cameras. The result of this will be the design of an AR headset that solves common problems in creating AR systems such as bulkiness, cost of manufacturing and dizziness, by providing a lightweight and low-cost solution.

The project will first create a differentiable renderer, then use it to design an optical part of aperture arrays and finally optimise the whole thing with modern machine learning libraries.

1.2. Aims and Objectives

The main aim of this project is to combine the optimisation advantages of modern-day python libraries with ray tracing and machine learning to stimulate an augmented reality interface with fast and high-quality rendering. The criteria for the success of this project can be broken down into these objectives:

- The first part involves constructing a ray tracing renderer that traces light rays from multiple light sources into an aperture array and then displays the resultant image on an image plane. This step is done to imitate the action of looking through a flat, rectangular-shaped component with differently sized holes on it. The effect of which should be seen on the rendered image.
- The resulting image from the ray tracing algorithm should be differentiable. This means that the gradient of the image must be computable because the system will pass the image gradient through an optimisation sequence that leads to improved results. This differentiability quality is to be achieved with the use of modern pythonic libraries that offer automatic differentiation.
- Selecting a suitable optimisation technique to improve rendering quality i.e., the rendered image is passed through a sequence that aims to render the image of the scene or the original image when looking through the aperture. The optimised result should be as close to the original image as possible and should be proved by taking the error difference between them into account.

Successful completion of these objectives will open the discussion for the use of coded apertures in the rendering and augmented reality field. The proposed system will offer the possibility of fast, efficient, and computationally cheaper rendering solutions to the AR industry.

Chapter 2: Literature Review

2.1. Differentiable Rendering

In computer graphics, rendering is the generation of two-dimensional or three-dimensional images by specifying the geometry, light, material, and light properties in a computer program [5]. The process is known to be complex due to its multiple parameters and high computational costs, thus much research has been conducted to simplify this process. One of which is the use of neural network pipelines. However, in order to use these neural network pipelines, the system must be differentiable, that is where differentiable rendering comes in.

Differentiable rendering computes the derivatives of the rendering function with respect to the scene parameters [6]. It allows for a 3-dimensional scene to be computed from a 2-dimensional scene. The differentiation of rendering functions is crucial for use of optimization techniques, neural networks and solving inverse problems. Most rendering techniques are not designed with differentiation in mind. For example with rasterization shifts in geometric parameters tend to be non-differentiable.

Ray tracing however offers a differentiable solution so far two conditions are met. One is that the rendering function being differentiated is continuous and the differential is continuous as well.

The earliest known instance of ray tracing was documented in 1525 by German painter Albrecht Durer who used an apparatus known as Durer's door to draw images [7]. The device uses a taut thread that passes through a hook on a wall on one end, which acts as the centre of projection and with a stylus attached to the other end. An assistant uses the stylus to move along the contours of the object being drawn.

The first computer-based ray tracing algorithm was done in 1968 by Arthur Appel [8]. His method focused on determining if a point is shaded or not by finding the closest surface to a camera at each image point. Another historical moment in ray tracing is in 1971 when a 30-second film of a helicopter and a gun undergoing various movements is made with ray tracing [9]. Modern ray tracing techniques have revolutionised the field of graphics to be able to generate images on computers and emulate their reactions to light i.e., shadows, and depth perception.

A differentiable ray tracing method is essential when designing optical devices as it can simulate the behaviour of optics through different mediums, hence allowing for parameter modifications before a physical prototype is built. The traditional approach to optical design is to focus light passing through a lens from the object plane unto a point spread function (PSF) disk on the image plane. Then by using the lens as a variable, we can compare the resulting PSFs of different lenses until a satisfactory image is obtained [10]. Modern optimisation techniques speed up this process significantly as all the lens and light parameters can be virtual, allowing for customisation.

A paper by ZongLing et al describes the use of a fast differentiable ray tracing (FDRT) technique in designing a single lens optical system [11]. Their method uses FDRT to get optical aberrations and then bases it on a convolutional neural network (Res-Unet) to recover the aberrated image. The optimisation target is to minimise the difference between a ground truth image and the restored image but also optimise the lens parameters. To make the rays differentiable, they are gaussianized meaning that the intersection of the ray with the image plane can be described by the gaussian distribution of the ray energy distribution.

Li et al [12] developed a differentiable renderer capable of handling secondary rendering effects i.e., shadows and global illumination. The method uses edge sampling of Dirac delta functions gotten from the scene differentials.

Using a differentiable layer is popular in the field as seen in Lui et al [13]. Here, they see rendering as an aggregate of the gradient contributions of triangles in a mesh and this allows for differentiation with rasterization and gives way to the method called soft rasteriser.

There also are programming interfaces created specifically for differential rendering. Some of these include OpenDR [14] known as an approximate differential renderer due to its rasterization limitations and RenderNet [15], a deep convolutional neural network for differential rendering.

2.2. Inverse Graphics

Inverse graphics aims to find the parameters that produce predefined realistic images. It is a top-down approach in which the images are used as the parameters to search for a model capable of producing said images. Many techniques in this field require derivatives in the rendering process.

Renders, however, are designed to be solved by the forward model of image generation which starts with parameters and ends up with a generated image. To combine inverse graphics with rendering, differentiable rendering is required. By setting the scene as the ground truth, the rendering algorithm provides an output, and a loss function is calculated. Through this, optimisation of specific parameters is possible to generate an output as close to the ground truth scene as possible. Hence, the parameters and the generated image are both expected outputs of our model.

An example of inverse graphics in rendering can be seen in [13, 16, 17] where the texture, light and geometry of an image are predicted with the use of neural networks by minimising the difference between the rendered image and the input image. These methods all require certain constraints to work e.g., multi-view images of the same object for training. In the paper by Zhang et al [18] they describe a system which runs on the idea of inverse graphics with renderers. A generative adversarial network (GAN) is a framework that generates new data based on previous data given to it and it is used in conjunction with the renderer to generate more images for optimising the renderer.

A paper by Zubic et al [19] describes an inverse graphics method that eliminates the use of rendering engines by evaluating an effective loss function that allows us to skip the rendering processes and go straight to the optimisation. It first uses a neural network to create a point cloud prediction of the 2-dimensional image in 3-dimensional space, then it computes the loss that tells how well the point cloud covers the true image and uses surface reconstruction to recover the 3-dimensional image from the point cloud.

2.3. Near Eye Displays

The basic components of a near-eye display (NED) involve an image source, magnifying optics and a medium to project virtual images to the viewers. With augmented reality near-eye displays (AR-NEDs), light from the real environment needs to also pass through. Most of the issues occurring in NED are design related such as the field of view, eye box size and resolution. There is also the issue of Vergence accommodating conflict (VAC) which occurs when there is a mismatch between the binocular disparity of a stereoscopic image and the single eye's optical focus cues [20]. This mismatch can cause headaches, nausea and general discomfort amongst NED users and is one of the biggest challenges facing the industry.

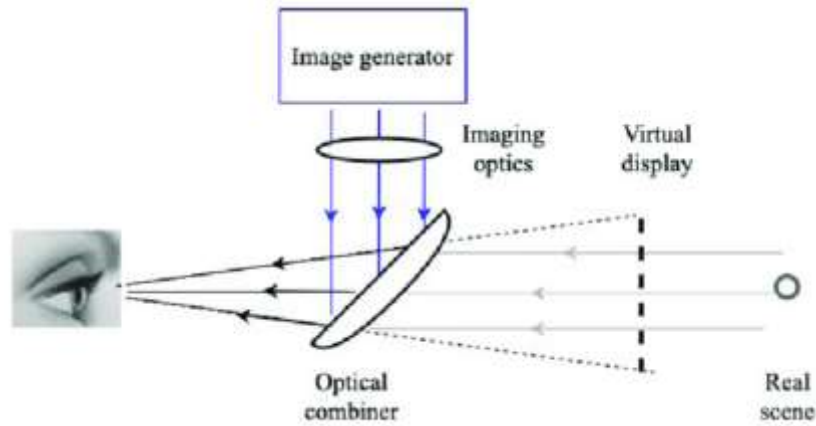


Figure 1: Main components of a near eye display [21]

2.3.1. Waveguide

Numerous techniques have been developed in the field to tackle the issues listed in section 2.3. One of which is the use of waveguides. A waveguide is an electromagnetic feed line and consists of a metal pipe [22]. In AR NED it is used to transmit virtual imagery into the users' eyes because it prevents loss of the input signal by total internal reflection. Two coupling components are required, one is the in-coupler which deflects the light from the image into the waveguide and the out-coupler is responsible for carrying the light to the eye [23]. Waveguide based near-eye display may use diffractive [24] [22] [25] or reflected [26] [27] optics techniques.

A common issue with this method is the limited field of view due to the incident light angle needed for total internal reflection. Solutions exist to address this in both reflective and diffractive techniques, but they come with a trade-off of eye box size and model complexity.

2.3.2. Holographic designs

Another popular method of implementing NEDs is holographic technology. A Holographic optical element (HOE) is an optical device that produces holograms with diffraction principles and has the unique property of wavefront manipulation. Recently, HOE's have become popular in augmented reality [28] because of the freedom in design and choice of materials.

Several optical design methods work with holographic technology [29] [30] [31]. Microsoft's HoloLens released in 2016 uses this technology for its mixed reality head-mounted display [32]. Meta also released a Virtual reality device [33] that works with holographic principles.

HOE-based near-eye displays are popular because the holographic elements can be manufactured in thin films, and they can achieve a large field of view [20].

However, a common issue with using HOEs is that it only reflects one wavelength of light, meaning to get a full RGB display three HOEs are required [2]. This leads to an increase in manufacturing and complexity costs. Furthermore, there is a trade-off between the eye box size and the field of view [34].

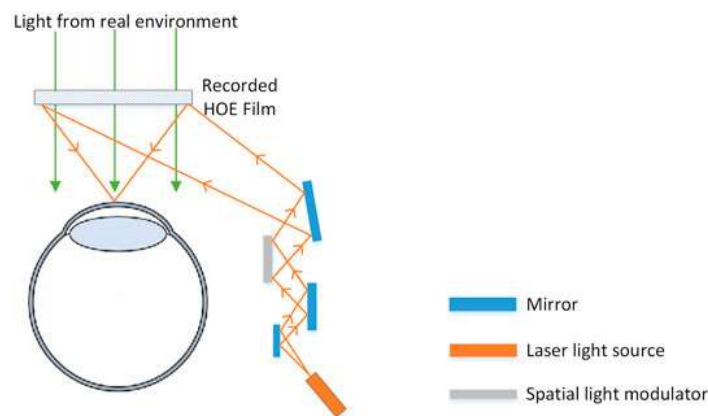


Figure 2:How Holographic displays work [20]

2.3.3. Pinholes

A recent technique in the field is the use of Pin lights. Maimone et al developed an augmented reality display in 2014, the concept uses an LCD panel and an array of point light sources placed directly in front of the eye [35]. The design has a wide field of view and saves manufacturing costs with the simplistic components. Another paper by Aksit et al [36] describes the use of pinhole apertures with mobile phones to create an augmented reality prototype. The use of pinholes is still fairly new to the augmented reality field, so there is room for more development.

2.4. Coded Apertures

Coded apertures are patterns of material opaque to various wavelengths of electromagnetic radiation i.e., gamma rays and x-rays [37]. Typically, the energy levels of gamma rays and x-rays are too high to be reflected straight away by lenses and mirrors, thus image modulation apertures block certain wavelengths and cast a shadow upon a plane. This shadow is reconstructed mathematically to retrieve the original radiation sources.

The pinhole camera is the simplest form of a modulation aperture. The advantage of pinholes over lenses is an infinite depth of field and lack of chromatic aberrations but its disadvantage is its low signal-to-noise ratio and throughput. This means it produces a blurry image with low resolution. To solve this, a mask of pinholes can be used and this, in turn, gives an imposition of multiple pinhole images that must be decoded to give a working solution.

A technique was created by Mertz and Young in 1967 to overcome the problems of using a single pinhole with celestial x-ray by creating the Fresnel zone plate (FZN) [38]. These plates consist of opaque and transparent circles with varying radii that are constructed in such a way that the x-rays passing through them are concentrated to a single point.

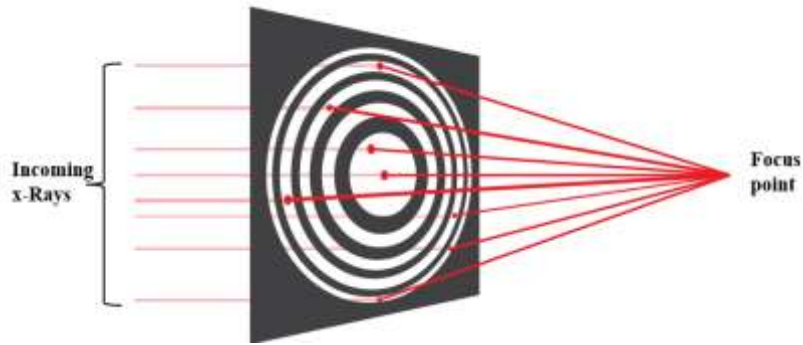


Figure 3: A Fresnel Zone Plate [39]

Figure 3 shows a five-ring Fresnel zone plate and the radius of the n th zone of the plate is defined as $r_n = r_1\sqrt{n}$, where r_1 is the radius of the innermost layer. The encoded image $r(x, y)$ is formed by equation 1, with $o(x, y)$ being the initial image, $a(x, y)$ as the aperture arrays and $*$ representing the two-dimensional spatial correlation.

$$r(x, y) = o(x, y) * a(x, y) \quad (1)$$

When decoding the encoded image, the aperture pattern is used as the decoding array $d(x, y)$. Thus, the decoded image $o'(x, y)$ is calculated as:

$$o'(x, y) = r(x, y) * d(x, y) = o(x, y) * [a(x, y) * d(x, y)] \quad (2)$$

The performance of an FZP increases with the number of rings as it has a binary transmission value and is finite in spatial extent.

Another coded aperture type is with random arrays of pinholes introduced in 1968 by R. H Dicke [40], who used the technique to form images of x-ray stars. The difference between random arrays and FZP is that the total open area is not equal to the closed area. However, just like FZP the size of the aperture is proportional to its performance. Gottlieb [38] [41] describes a method in which the aperture properties vary with time and the encoded result is a function of time. By choosing a specified aperture function, a delta function to decode the resulting image can be gotten.

Coded apertures have mostly been used in x-ray and gamma-ray imaging systems but have recently become a researched field for near-eye displays. Aksit et al describe a near-eye display system using an array of pinholes for virtual reality applications [36]. In their method, the coded aperture is an array of pinholes, and the initial image is a mobile device screen. Other methods [42, 43, 44] also exist, each with clearly defined apertures for near-eye displays.

A reoccurring issue with the use of pinholes as the coded aperture is overlapping images which can lead to visual fatigue and discomfort in a user.

2.5. Optimisation

Optimisation deals with finding the best possible solution to a problem with multiple answers. The concept has existed since 100 B.C when evaluating the shortest distance between two points but the mathematical method can be dated to the 17th century when Fermat and Newton investigated the way to find the optimum of a single variable [45]. Their hypothesis proved that the minimum point of a function will lie in the area where the derivative is zero.

However, not all functions have singular variables. This means that the function would require more complicated mathematical methods to find the minimum of each variable. Additionally, the solution to the optimisation problem may be specified i.e., constrained optimisation. It is for these reasons that optimisation is such a well-researched field of study, applicable across industries.

In computer graphics, optimisation aims for improved rendering quality and faster rendering engines. Most optimisation methods in ray tracing involve reducing the number of rays generated or reducing the number of objects each ray must be tested against in the scene.

2.5.1. Algorithms to reduce the number of rays

Anti-aliasing of rendered images is one of the most common ways to improve their quality. The images rendered may have pixelated or jagged edges due to being shown at a lower resolution than it is originally. A popular anti-aliasing method is supersampling. It aims to reduce image defects by rendering the image at a higher resolution than it is originally, and then shrinking it to its required size. Adaptive super sampling is an improved version of this that sends multiple rays to a pixel and compares their colour, if there is no colour variation then it is assumed to be a single object and the average of all rays is computed [46]. Anti-aliasing can cut down the number of rays needed to render an image as it finds situations where a small number of rays lead to accurate results.

The very first technique to reduce the number of rays was Hall et al's adaptive tree depth control in 1983 [47]. It works by thresholding the resulting colour generated by the rays and if the colour falls below that threshold then that ray and its effect are ignored.

2.5.2. Algorithms to reduce the number of objects tested against each ray

A method to reduce the number of tests to check if a ray hits an object is pre-processing the model by separating the scene into different volumes. A rank is assigned to each and then the test is carried out only for the volumes with higher rankings. This can reduce the number of rays-to-object tests. An example is a method created by Sweeney et al for B-spline surfaces [48].

Another method is using Z-buffer algorithms. A Z-buffer is used in graphics to represent the depth information of objects in a 3D space and is used to determine if the object is visible or being obstructed. Objects which are found to be visible can then be prioritised, hence reducing the objects tested against each ray.

In recent research, some techniques seek to optimise the rendering result instead. These methods may include an optimised ray tracing algorithm and when the image is produced that image is further refined through artificial intelligence processes.

2.6. Artificial Intelligence and Ray Tracing

Artificial intelligence (AI) is the simulation of human intelligence processes with the use of computers. Machine learning is a branch of AI that can produce new data based on previous information. It is a computer-based method that learns from old processes to give new and better results thus, modelling the human method of learning.

Neural networks are computer-simulated models of the human brain which are expected to learn and produce new information based on what they are given. It carries out the machine learning process for multiple layers, whereas each new layer returns a result that is used to generate an even better result. It is a large, interconnected, information processing system built to solve a specific task. The first instance of an artificial neural network was in 1943 by McCulloch and Pitts in a paper titled “The Logical Calculus of the Ideas Immanent in Nervous Activity” [49].

Since that paper, there have been great strides in the field of research for artificial neural networks and they have been used in a wide range of fields. The main uses are for predicting data based on existing information, finding patterns or sequences in data and data processing operations. This research focuses on the data processing aspect, specifically image processing. In a review on image segmentation techniques, it was predicted neural networks would gain popularity within image processing [50] and this prediction turned out to be correct.

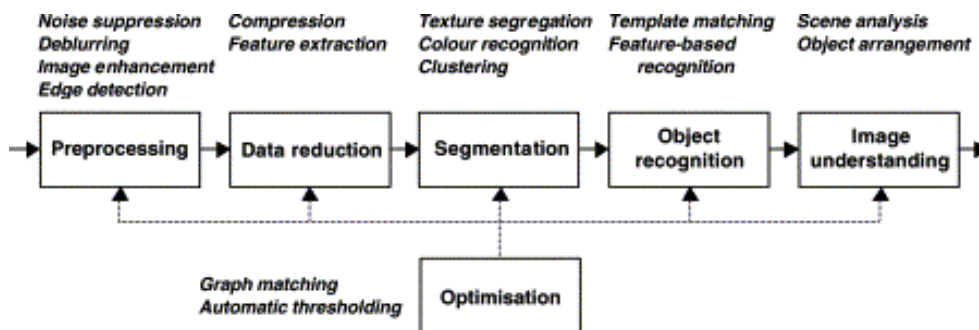


Figure 4: Different types of image processing with neural networks [51]

The different uses within the image processing field can be seen in figure 4 and they range from segmentation to object recognition [52]. In each of these procedures, optimization is taken as a necessary part of the process.

The main components of them include an input layer, a processing layer, and an output layer. Additionally, the parameters to be defined are the patterns of interconnections between the layers, the weights and an activation function that controls the value of the output.

The most common way of applying Artificial intelligence to graphics is by performing denoising or transformation on the rendered image to generate a higher quality or modified image. This can be seen in [53, 54, 55] where the rendered image is passed through a convolutional autoencoder for denoising. Raytracing has also been used to generate synthetic training data for neural networks instead of using real pictures [56]. Technology companies like Nvidia are already combining the capabilities of ray tracing and neural networks to produce high-quality realistic rendering [57].

Chapter 3: Theory

3.1. Ray Tracing

The pinhole camera as seen in figure 5(a) consists of a box with a pierced hole at its front, which is covered when not in use and placed in a dark room. At the back of the box, photographic film is placed to capture the information that light rays deliver.

When the inside of the box is exposed to light i.e., the covering is removed from the hole, incoming light rays will strike the photographic film and cause a chemical reaction that captures the information delivered by the light source to the film. The covering is then replaced.

Using a small pinhole ensures that only a limited number of light rays pass from the environment to the film, and this is because the number of light rays passing through is directly proportional to the image brightness and inversely proportional to its clarity. In the scenario where the pinhole is absent and the film is exposed directly to all the light rays in the environment, the resultant image would be blank.

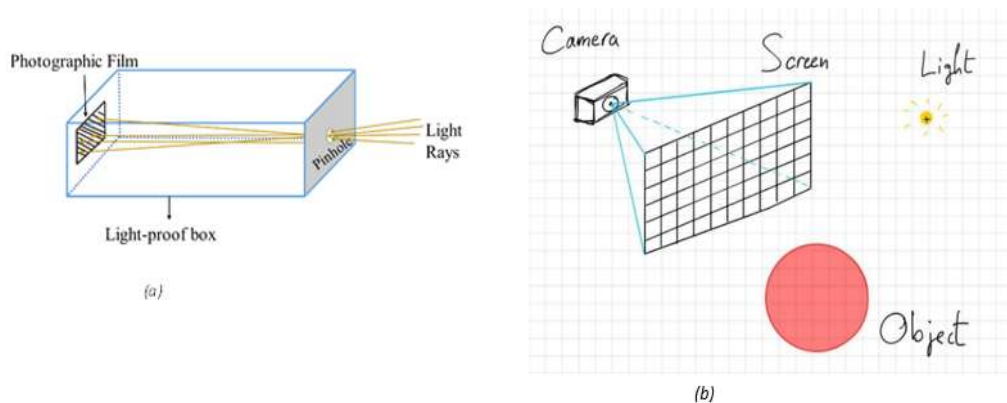


Figure 5: (a) The Pinhole camera model, (b) Computer Graphics Pinhole model [58]

In computer graphics, the camera is simulated by placing the film plane in front of the pinhole and the pinhole is renamed to *Camera* as seen in figure 5(b) but is also referred to as the eye. Therefore, the camera can only detect objects in front of the screen and if the object is larger than the boundaries of the screen it is clipped.

Generating the image on the screen means assigning each pixel on a plane to a colour. A ray of light is the pathway a light particle takes as it travels through space and when the particle

strikes a pixel on the image plane, its energy level determines what colour it is detected as. As many light rays may strike the same pixel, each would all have its colour, so to assign a single colour to the pixel all the available colours would be averaged. For example, if a red particle and a blue particle arrive at the same pixel, we will perceive the average of the colours, purple.

Following the path of the light particle as it interacts with the scene is the principle of ray tracing. The light particle will start from the source e.g., the sun or a lamp and bounce off objects in the scene, affecting its brightness level before returning to the image plane where it is added with other light particles and assigned to a pixel.

Tracing the path from the light source to the scene is known as *forward ray tracing*. It can accurately detect the colour of each light particle; however, it is known to be highly inefficient and very slow. This is because there are a good number of light rays that never interact with the scene and pass straight through as well as light rays that hit objects that are not within the screen boundaries. Using forward ray tracing, every light ray would be traced leading to a lengthy, time-wasting and computationally expensive process that may not yield any result.

Reversing the *forward ray tracing* technique is a more efficient idea. This means the ray is traced from the camera to the nearest object it hits and is known as *backwards ray tracing*. So only the light rays that are sure to interact with the scene are traced. In graphics, *backwards ray tracing* is almost exclusively used.

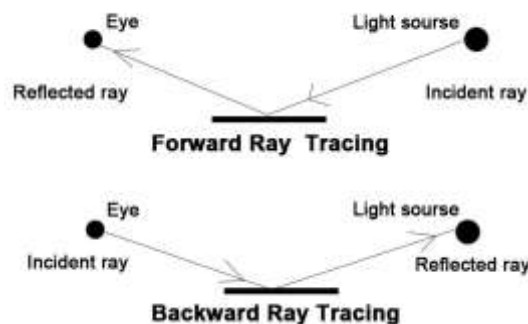


Figure 6: Ray Tracing techniques, forward ray tracing and backwards ray tracing [59]

Light rays can further be divided into four separate categories. The first is pixel rays which carry light particles from the pixel to the camera, then reflected rays that carry the light reflected by objects in the scene, illumination rays carry light from the light source to the object surface and finally transparency rays that pass light through objects.

The rendering equation describes how light on the object behaves. It is a combination of reflected and emitted light on the object represented as $L_o(x, w_o, \lambda, t)$ and is a function of the location is space x , the light wavelength λ , time t and the direction of the outgoing light, w_o in equation 3 below. The emitted light is described by L_e and reflected light is the integral of the dot product of the bidirectional reflectance distribution function f_r , the incoming light L_i and the dot product of the surface normal n and direction of incoming light w_i :

$$L_o(x, w_o, \lambda, t) = L_e(x, w_o, \lambda, t) + \int_{\Omega} f_r(x, w_i, w_o, \lambda, t) L_i(x, w_i, \lambda, t) (w_i \cdot n) dw_i \quad (3)$$

The bidirectional reflectance distribution function (BRDF) f_r describes the behaviour of reflected light on an opaque surface and is defined as the ratio of the radiance, which is the power of a photon per unit solid angle, per unit projected area $L_r(w_r)$, to scattered irradiance $E_i(w_i)$ i.e., power of a photon per unit projected area. The angle θ_i is the angle between the incident ray direction, w_i and the normal, n .

$$f_r(x, w_i, w_o, \lambda, t) = \frac{dL_r(w_r)}{dE_i(w_i)} = \frac{dL_r(w_r)}{L_i(w_i) \cos \theta_i dw_i} \quad (4)$$

3.1.1. The intersection of a ray with a plane

Firstly, a planar surface is defined by four vectors A, B, C, D , where ABC are its normal P_n coordinates and D is the distance between the coordinate system origin to the plane:

$$P_n = P_{normal} = [ABC] \quad (5)$$

$$Plane = A * x + B * y + C * z + D$$

$$\text{Where, } A^2 + B^2 + C^2 = 1 \quad (6)$$

Then, a ray is defined in terms of its origin vector, R_0 i.e., the eye and its normalised direction vector, R_d represented by three-dimensional coordinates. The intersection points between the ray and the planar surface, R are given by equation 7, where t is the distance from the origin to the intersection point.

$$R_0 = [X_0, Y_0, Z_0] \quad (7)$$

$$R_d = [X_d, Y_d, Z_d] \quad (8)$$

$$R(t) = R_0 + R_d t = [x, y, z] \quad (9)$$

The value of t needs to be evaluated to find the intersection points and this can be done by substituting the ray equation into the plane equation and solving for t . This gives us the ratio of the plane's normal P_n multiplied by the ray origin R_0 , plus the distance from the plane to the origin D and the plane normal P_n , multiplied by the ray direction R_d .

$$t = -\frac{-(A * X_0 + B * Y_0 + C * Z_0 + D)}{A * X_d + B * Y_d + C * Z_d} \quad (10)$$

$$t = -\frac{P_n R_0 + D}{P_n R_d} = \frac{v_0}{v_d} \quad (11)$$

In the case that $v_d = 0$ in equation 11, then the ray does not intersect with the planar surface and passes straight through. Therefore, $v_d > 0$ is a condition that must be satisfied for an intersection to occur. Another condition is that the intersection point t must not be less than zeros or have a negative value as it means that the ray intersects behind the plane and thus is not visible.

When the ray intersects the plane, it may land in between two pixels, there is then a question of which pixel that intersection point should be assigned to. This is solved by considering the nearest neighbouring pixel of the intersection point. Firstly, the distance between the intersection points and all pixels is calculated, then the minimum distance is selected and the pixel with that minimum distance is assigned to the intersection point t .

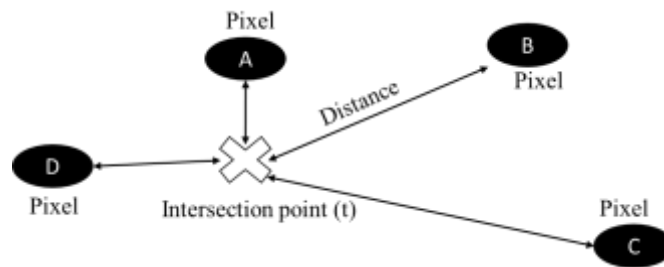


Figure 7: The distance between the intersection points and the surrounding pixels. In this image, the intersection point would be assigned to pixel A, as it has the smallest distance between them.

An additional condition states that the minimum distance cannot be larger than the distance between the pixels. This is to ensure that the assigning of pixels to points remains within the image boundaries. Through the assigning of pixels to points, an image can be generated through ray tracing.

3.2. Optimisation

In its simplest form, an optimisation problem can be solved by minimising or maximizing an objective function. The object function or cost function is the mathematical expression of the objective to be achieved by the program, this can be reducing expense, increasing profit, reducing the time taken, improving material quality etc. The function allows for the comparison of different variables to determine which is the best. The process of identifying the objective function, variables and constraints is known as modelling and is the first step taken to define an optimisation problem. Modelling can be a complicated process, as if the problem is too simplified or too complicated, then it would be an inaccurate representation of the problem, leading to inaccurate solutions.

In the case where the objective function can be represented graphically by a straight line, it is called *linear programming* and the objective function is a linear function. Meanwhile, *non-linear programming* or *dynamic programming* solves non-linear functions by breaking them down into simplified units.

Optimisation problems can also be divided based on the existence or absence of constraints, which are requirements that the solution must satisfy to be accepted. For example, if optimising the quality of materials in a manufacturing plant, the constraint is the standard the material needs to be to be accepted.

3.2.1. Constrained Optimisation

These are problems with specifications on the solutions. They are known to be more complex than unconstrained problems as the more constraints a function has, the more difficult it is to create a streamlined solution that satisfies all the requirements. Computationally, a constraint would be a stopping condition of the minimisation algorithm e.g., if we want the minimum to be no less than 10 the algorithm would run for values of $x > 10$.

3.2.2. Unconstrained optimisation

Unconstrained optimisation problems have no restrictions on acceptable solutions. They consist only of the objective function and are written as:

$$\min f(x) \quad (12)$$

Where f is the objective function and x is the variable that allows for the decrease of the objective function and is known as the optimisation variable. In an algorithm, the value of x will be changed several times until the minimum of $f(x)$ is achieved.

For a point to be considered a minimum, it must fulfil some conditions. Firstly, the gradient value at the minimum must be zero. The gradient of a function is a vector that shows the rate of change of that function and to get to the minimum where the gradient is zero, an algorithm called gradient descent is used.

The second condition is that its eigenvalues are positive definite. An eigenvector is a vector that does not change direction in the event of a transformation and the eigenvalue is the amount the vector changes by. It is determined by the equation:

$$Av = \lambda v \quad (13)$$

$$|A - \lambda I| = 0 \quad (14)$$

Where λ is the eigenvalue, A is the matrix form of the second derivative of a function and v is the eigenvector. Finding the eigenvalues is done by solving the resultant equation from finding the determinant. When the eigenvalues are positive definite, it means they are positive and not equal to zero. There are methods of finding the minimum of a function based on the value of the hessian which is a matrix of the second-order partial derivatives of a function.

3.2.3. Gradient descent

Gradient descent is a first-order optimisation algorithm used to find the minimum of a function. It can be said to be the basis of machine learning as it aims to minimise the loss of a function. However, the function must be differentiable and convex to be minimised by gradient descent. A differentiable function is one in which its derivative exists at each point in its domain and a convex function means that the line segment connecting two points on the

function lies above its curve. Figure 8 shows the graph of a function that is differentiable and convex.

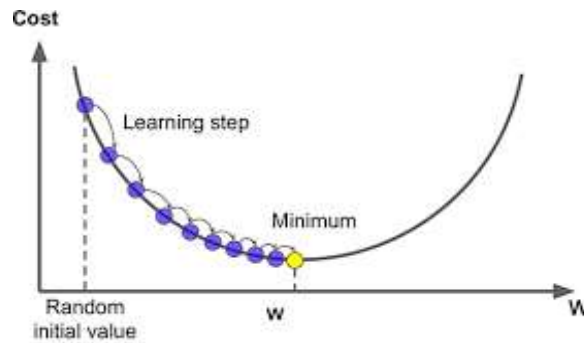


Figure 8: Visualisation of a convex and differentiable function undergoing gradient descent [60]

Gradient descent calculates the minimum by taking steps proportional to the negative gradient of the function at a point. Suppose the convex function is visualised as a bowl with ridges on the inside and the minimum of the function is at the bottom. Calculating the gradient at a ridge would point the direction towards the top of the bowl, so to move towards the bottom we take the negative gradient. We move towards the direction of this negative gradient, stop at a new ridge, calculate the gradient at the new ridge then move again. Doing this process multiple times should eventually lead to the ridge at the bottom of the bowl where the minimum lies, and this is the process on which the gradient descent algorithm operates. The mathematical representation of gradient descent is given as:

$$p_{n+1} = p_n - \eta \nabla f(p_n) \quad (15)$$

Where $\nabla f(p_n)$ is the gradient of the function at a point p_n , p_{n+1} is the newly calculated point and η is the step size or learning rate that controls how much the function moves in the direction of the negative gradient. The step size parameter is very important as it can determine whether the minimum is reached or not. If the step size is too large, then the algorithm may overshoot and land at a point far away from the minimum. Alternatively, if the step size is too small, the algorithm may move too slowly.

Gradient descent can have issues navigating the area around the minimum as this area could have ravines i.e., areas where the surface curves more steeply. This leads to oscillation around the slopes of the ravine, making it difficult to reach the minimum. Momentum is a technique that accelerates the descent around this area and reduces oscillations. It is done by

adding a momentum parameter γ that is the fraction of the past time step update vector to the current update vector. The updated equation is:

$$p_{n+1} = \gamma p_n - \eta \nabla f(p_n) \quad (16)$$

The value γ is usually set to 0.9 and leads to faster convergence. Another way of speeding up the descent process is through an Adaptive gradient. It works by customising the learning rate to the parameters based on previous observations i.e., it uses a different learning rate for each parameter. Each parameter is represented as θ_i and the gradient of parameter θ_i at timestep n is denoted as $g_{n,i}$. The symbol G_n represents a diagonal matrix of the squared sum of all the past gradients and ϵ is a smoothening term. The mathematical representation of adaptive gradient is given as:

$$\theta_{n+1,i} = \theta_{n,i} - \frac{\eta}{\sqrt{G_n + \epsilon}} \cdot g_{n,i} \quad (17)$$

Additionally, there are specialised forms of adaptive gradient that follow the same principle but have a different implementation. Adadelta was created to solve issues of Adaptive gradient by restricting the collection of past gradients to a fixed size. Rather than calculating the squared sum of all past gradients, it calculates the decaying average of all past squared gradients and the G_n term is replaced by a diagonal matrix containing this decaying average $E[g^2]_n$.

$$E[g^2]_{n+1} = \gamma E[g^2]_n + (1 - \gamma) g_n^2 \quad (18)$$

$$\theta_{n+1,i} = \theta_{n,i} - \frac{\eta}{\sqrt{E[g^2]_n + \epsilon}} \cdot g_{n,i} \quad (19)$$

Adaptive Moment Estimation (Adam) can be said to be the most popular optimiser used in optimisation. It is an adaptive gradient method, but it also uses an exponential of decayed averages of past gradients m_n in its calculation. Firstly, the decaying average of past gradients m_n and past squared gradients v_n are calculated:

$$m_n = \beta_1 m_{n-1} + (1 - \beta_1) g_n \quad (20)$$

$$v_n = \beta_2 v_{n-1} + (1 - \beta_2) g_t^2 \quad (21)$$

β_1 and β_2 are decay rates usually set as 0.999 and 10×10^{-3} . The values of m_n and v_n are also called the mean and the variance. Next, these values are recalculated to avoid bias as they are biased towards zero and put into the new update equation for each parameter:

$$\widehat{m}_n = \frac{m_n}{1 - \beta_1^n} \quad (22)$$

$$\widehat{v}_n = \frac{v_n}{1 - \beta_2^n} \quad (23)$$

$$\theta_{n+1,i} = \theta_{n,i} - \frac{\eta}{\sqrt{\widehat{v}_n} + \epsilon} \cdot \widehat{m}_n \quad (24)$$

Other methods of adaptive gradients include AMSGrad, RMSprop and Nesterov-accelerated adaptive moment gradient (Nadam) [61]. There are also more optimisers still in development, which then begs the question of what the best optimiser to use is. It all depends on the model to be optimised and the level of data available. In instances where the data is limited, the adaptive gradients method is said to lead to better results and the learning rate does not need to be fine-tuned.

Nevertheless, the original gradient descent method is still being used as an optimiser and can give good results. In scenarios where fast convergence is not a priority and the model is not overly complex, the gradient descent algorithm is expected to be successful.

Chapter 4: Design



Figure 9: Flow chart of system design

4.1. Image-pre-processing

The algorithm starts by pre-processing the image by normalising it and reshaping it to a smaller size. Image normalisation means changing the range of pixel intensity values from a range of 0 to 255 to a range of 0 to 1. It is done to speed up the computation process. The normalised image is then converted into a tensor which is a multidimensional array that holds all the pixel properties of the image and reshapes to the same size as the aperture. The images used for testing and processing are retrieved from the Modified National Institute of Standards and Technology (MNIST) database [62], test images from the UCL's image processing module [63] and independently generated images from a Pixel art [64] website.

4.2. Initialising the light sources

The light source location is defined as a tensor grid of values. As the system simulates an aperture in the real world, it assumes that light comes from natural sources, and this means there will be an abundance of light coming from different directions. However, as the only light rays required are the array of rays which pass through the pinholes, the size of the light source positions is set to the same size as the aperture array.

4.3. The Aperture Array

The coded aperture used in the algorithm is modelled after that used in the paper by Aksit et al [36], where the aperture is a rectangular surface with holes at different points. Each of these holes allows for light to pass through them and onto the detector surface, while the opaque parts do not allow light. Thus, points on the array can be labelled with numbers, ranging from 0 to 1, where 0 represents complete opacity and 1 means it allows for light to pass through. The aperture generated for grayscale images is a 28cm by 28cm aperture gotten through the *torch.rand()* method from the PyTorch library which is described in detail in

section 3.7.1. It generates random numbers ranging from 0 to 1 and is made differentiable by setting the *requires_grad* parameter to *True*. The array is placed 1cm away from the image source.

For the case of coloured images, a 28 by 28 by 3 random array is generated instead. The third dimension represents randomly generated RGB (Red, green, and blue) channels of an image.

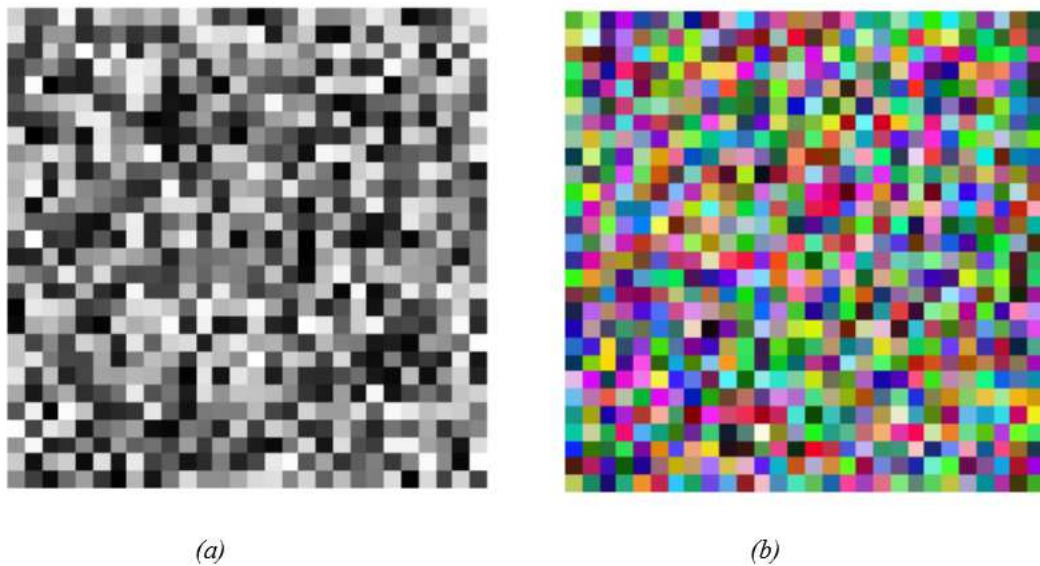


Figure 10: (a) 28 x 28 Aperture Array for grayscale images with values ranging from 0 to 1, (b) 28 x 28 x 3 array for coloured images.

4.4. The Forward

The forward in machine learning refers to the calculation aspect that gives an output from a specific set of inputs. It is this output that is used in calculating the loss in an optimisation sequence. Here the forward contains the ray tracing aspect that is used to calculate the intersection point of rays from the light source to the image plane and generate the resultant image. The function returns the image seen when looking through the aperture and a list of the intersection points of the rays with the image plane.

4.5. Selecting the loss function

The cost function is the squared difference between the prediction and the target value. In this case, the predicted value is that of the rendered image and the target is the input image. It is also known as the L2 loss and is mathematically represented as:

$$Loss = \sum_{i=1}^n (y - \hat{y}_i)^2 \quad (25)$$

Where y is the target (input) image and \hat{y} is the predicted (rendered) image. In PyTorch it is calculated by the module *torch.nn.MSELoss()*.

4.6. Optimisation

The optimisation technique chosen is stochastic gradient descent for of its simplicity and because only a single parameter is being optimised i.e. the aperture array. Initialising the optimisation step is done with torch vision's *torch.optim.SGD()* module which allows for step size, momentum and optimisation parameters to be specified. Selecting the step size is done by trial and error Eventually, a value of 0.5 is picked. The momentum value is also set to 0.9. The optimisation step is run for 300 epochs in which each epoch updates the aperture array by taking the loss of the output image and the input image.

4.7. Python Libraries

4.7.1. Graphical processing units (GPUs) and PyTorch

Graphical processing units (GPU) were designed to improve the capacity of rendering 3D graphics in games. The first GPU was the “GeForce 256” and was created by the company NVIDIA in 1999 [65]. Before its invention, graphics in computers were handled by video graphic arrays (VGA), consisting of DRAM (Dynamic random-access memory) and a display generator.

Over time, GPUs have become better at creating very realistic rendering and are more generally used for high computing workloads, deep learning and more. They consist of many energy-efficient computational cores with large memories, meaning they can perform several computations simultaneously. A GPU can come attached to a computer's central processing unit (integrated) or as a separate add-on chip that can be mounted on a circuit board (discrete). Some popular GPUs include NVIDIA's CUDA, Intel's UHD and AMD's Radeon.

PyTorch is an open-source machine learning library used specifically for python-made applications in computer vision, deep learning and natural language processing developed in 2016 by Meta AI [66]. It is based on a previously existing library Torch, which was written in

Lua programming language. However, it did not gain popularity as researchers were reluctant to learn the Lua language to use the library. This led to the development of PyTorch which uses the familiar python programming language.

PyTorch is widely used in deep learning as it offers the advantage of supporting GPUs and automatic differentiation, thus speeding up the deep learning process significantly. It is also easy to debug as it is based on the popular programming language, python. CUDA is the most common GPU used alongside PyTorch. By combining the GPU advantage of faster graphics rendering and the PyTorch advantage of improved machine learning capabilities, a deep learning-based rendering technique can be created.

4.7.2. ODAK

The Odak library, created by the computational light laboratory of the University College London is a toolkit for computer graphics, visual perception, and optical sciences [67]. It contains essential functions for ray tracing used in this project namely:

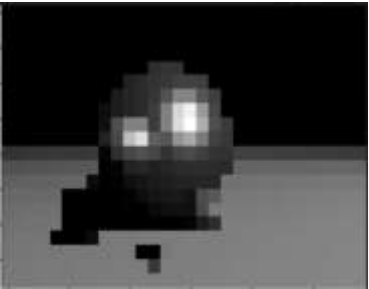
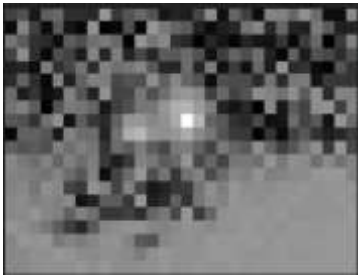

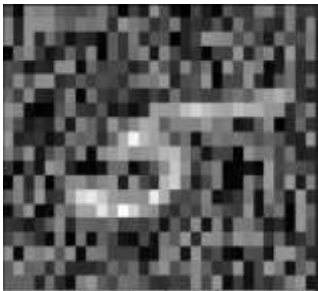

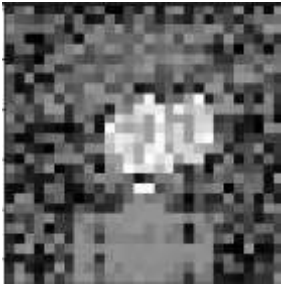
- ***Odak.raytracing.create_ray_from_two_points (ray_origin, ray_end)***: Takes XYZ definitions of a ray's starting and ending point and returns an array defining the ray
- ***Odak.raytracing.define_plane (point)***: Takes an XYZ point defining the centre of the plane and returns points defining a planar surface
- ***Odak.raytracing.intersect_w_surface (ray, points)***: Takes in the definition of a ray and a planar surface and returns an array containing the surface normal at the intersection point and the distance from the ray origin to the intersection point on the planar surface.

4.7.3. Image processing libraries (OpenCV, Matplotlib, Pillow)

The project also makes use of various image processing libraries for image retrieval, pre-processing, post-processing, and visualisation. The OpenCV and Pillow libraries offer options for reading images, writing images, conversion to greyscale and resizing before being placed in the ray tracing model. Matplotlib is a popular python visualisation library and is how the input and rendered image are seen. However, it can only display images defined as arrays, specifically Numerical Python (NumPy) arrays, so PyTorch tensor arrays to NumPy arrays conversions are needed.

Chapter 5: Results and Analysis

5.1. Initial results with a randomised array

	Test Image	Ray Tracing Through Randomised Aperture	Loss
1			1.79×10^{-1}
2			3.02×10^{-1}
3			2.44×10^{-1}









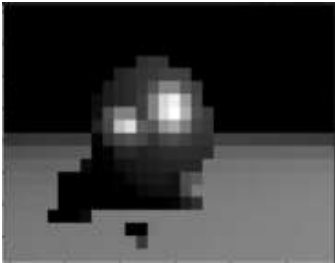
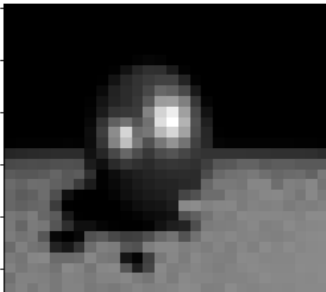
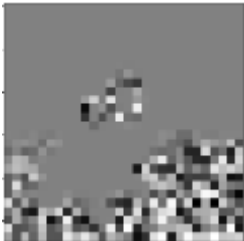


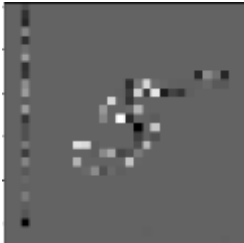


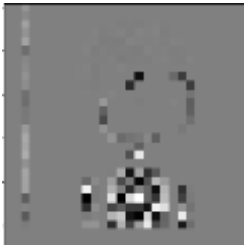


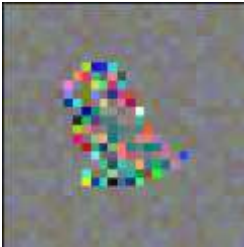
4			1.45×10^{-1}
5			1.32×10^{-1}
6			2.06×10^{-1}
7			1.1×10^{-1}
8			4.76×10^{-2}

Table 1: Result of Ray Tracing through randomised aperture array

5.2. Optimisation Results

	Test Image	Ray Tracing Through Optimised Aperture	Optimised Array	Loss
1				4.0×10^{-3}
2				3.0×10^{-4}
3				3.0×10^{-4}
4				1.9×10^{-3}


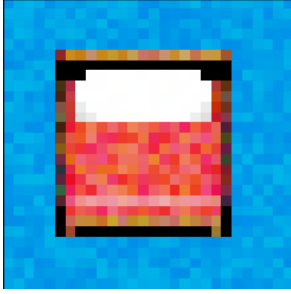
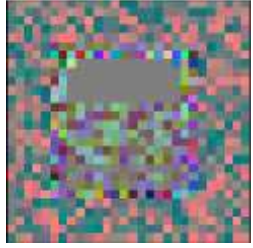


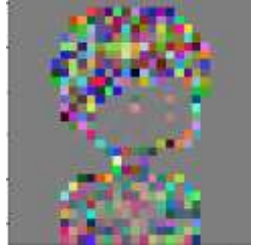


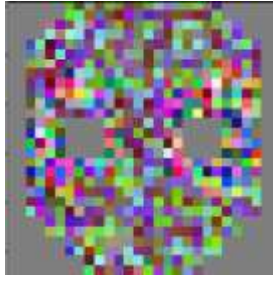


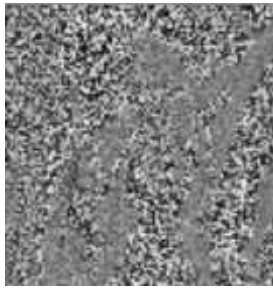
5				2.2×10^{-3}
6				2.1×10^{-3}
7				2.5×10^{-3}
8				8.8×10^{-3}

Table 2: Result of Ray Tracing through optimised aperture array

5.3. Percentage reduction in MSE loss

Tests	Percentage (%)
1	97.0
2	99.9
3	99.9
4	98.6

5	98.3
6	98.9
7	97.7
8	81.5

Table 3: Percentage reduction in losses

5.4. Analysis

Experimentation is carried out for 4 grayscale images and 4 colour images. The images are put through the ray tracing algorithm to give a simulation of looking through a randomly generated array and then through an optimised version of the array. The loss value in the first instance is on average within the 10^{-1} range while in the second instance with the optimised aperture the loss values decrease to values between 10^{-3} and 10^{-4} . The loss value for all test cases decreased by an average of 96.5%.

The optimised aperture is also plotted alongside the images in table 2 and from inspection, it can be said to be a mask of the optimised image. The dark areas in the images are represented by blocked-out areas in the optimised aperture array with a value of 0, as light is not allowed through these areas. The areas of the optimised image with details allow light to pass through and are represented by pixels that show the intensity of the passing light. Thus, this shows the expected results from the algorithm as the array has been modified such that when looking through it the target image is seen.

The ray tracer takes 0.72 seconds to generate a solution for colour images and 0.34 seconds for grayscale images. Each test example converges in 300 steps to a solution and took about 5 minutes to complete for a 28 by 28-pixel image.

For the 8th test case, the dimensions of the image and aperture array are changed to a 100 by 100cm grid to show the effect of running the algorithm on a larger scale. Ray tracing for this new scale took 3 seconds while the optimisation step was run for twice as many epochs i.e., 600, with a step size of 0.8 and in total took an hour but was still able to provide a solution with an 81.5% decrease in loss.

Chapter 6: Conclusion and Further Improvements

This project set out to explore the concept of differentiable ray tracing achieved through modern machine learning libraries in designing an optical part. It began by detailing existing rendering concepts and methods of attaining differentiability, then divulged into the field of machine learning and its history with ray tracing rendering methods. The system focused on designing the optical part as an augmented reality interface and did so by creating a component that serves as the AR headset. The component is a square that is opaque except for several pinholes on it, through which light is allowed through. The arrangement and amount of light that passes through these holes on the components are initially randomly generated. A simulation of looking through the randomly generated aperture is generated by ray tracing an image through the part and it gives a scrambled version of the original image.

Afterwards, the original image and the scrambled rendered image go through an optimisation sequence to modify the component such that the pinholes and light amplitudes are positioned in such a way that looking through the component shows the original image. Results show that the difference between the original image and the rendered image of looking through the optimised headset is within the 10^{-3} to 10^{-4} range with a 96.5 % decrease in the loss values of looking through the component with randomly placed pinholes.

These results are promising and give an insight into what can be achieved in the AR field with differentiable ray tracing and opens up discussions in designing optical parts with the technique. The same technique of differentiable ray tracing could easily be applied in designing other optical components e.g., lens design.

In the initial plans shown in section 8.3. of the appendix, a physical model of the optimised aperture was to be created through 3d printers. However, time constraints did not allow for it, thus it is a potential continuation of the project. The holes on the aperture will be sized according to the light amplitude values of the aperture. This is fitted on an image source to give a cheap, lightweight augmented reality display. Alternatively, the AR display could contain all the steps of this system in a single processing unit, so it can capture images, and ray trace, and optimise in real-time.

Additionally, optical effects can be simulated with this method. For example, the target image could be set to an enlarged image and through the system, the image target image can be reached to produce an enlarged image when looking through the aperture. This effect may also be achieved by using the distance from the aperture to the image source as a parameter to be optimised.

Comparing the method with existing rendering engines, the ray tracer seems to converge to a solution very quickly. However, the speed can still be improved with the ray tracing-specific optimisation techniques discussed in section 2.5 An example of this is performing importance sampling on the light source. This would ensure that only the light rays with a high probability of hitting the surface are computed. An even faster ray tracer would mean that the algorithm can be executed for larger-sized images and videos with numerous frames per second.

Chapter 7: Bibliography

- [1] Research and Markets, “The Worldwide Near-Eye Display Industry is Expected to Reach \$5.3 Billion by 2027,” Research and Markets, 30 May 2022. [Online]. Available: <https://www.prnewswire.com/news-releases/the-worldwide-near-eye-display-industry-is-expected-to-reach-5-3-billion-by-2027--301557290.html>. [Accessed 08 June 2022].
- [2] X. Xia, F. Y. Guan, Y. Cai and N. M. Thalman, “Challenges and Advancements for AR Optical See-Through Near-Eye Displays: A Review,” *Frontiers in Virtual reality*, 2022.
- [3] A. L. d. Santos, “Real Time Ray Tracing for Augmented Reality,” in *Virtual and Augmented Reality (SVR)*, Recife, Brazil, 2014.
- [4] A. L. d. Santos, D. Lemos, J. E. F. Lindoso and V. Teichrieb, “Real Time Ray Tracing for Augmented Reality,” in *Symposium on Virtual and Augmented Reality*, Brazil, 2012.
- [5] Wikipedia, “Rendering (computer graphics),” [Online]. Available: [https://en.wikipedia.org/wiki/Rendering_\(computer_graphics\)](https://en.wikipedia.org/wiki/Rendering_(computer_graphics)). [Accessed 10 09 2022].
- [6] A. Robineau, “An overview of Differentiable Rendering,” Qarnot, 18 October 2021. [Online]. Available: <https://blog.qarnot.com/an-overview-of-differentiable-rendering/>. [Accessed 14 June 2022].
- [7] G. R. Hofmann, “Who invented ray tracing?,” *The Visual Computer*, p. 120–124, 1990.
- [8] A. Appel, “Some techniques for shading machine rendering of solids,” *IBM Research center*, 1968.
- [9] G. Robert and N. Roger, “3-D Visual Simulation,” *Simulation*, pp. 25-31, 1971.
- [10] T. V. M. D. B. L. G. M. A. W. N. B. A. S. E. F. Andrew R. Harvey, “Digital image processing as an integral component of optical design,” in *The international society for optical engineering*, 2008.
- [11] Z. Li, Q. Hou, Z. Wang, F. Tan, J. Liu and W. Zhang, “End-to-end learned single lens design using fast differentiable ray tracing,” *Optics Letters*, vol. 46, no. 21, p. 5453, 2021.
- [12] M. A. F. D. a. J. L. Tzu-Mao Li, “Differentiable Monte Carlo Ray Tracing through Edge Sampling,” *ACM Transactions on Graphics*, vol. 37, no. 6, 2018.

- [13] T. L. W. C. H. L. Shichen Liu, "Soft Rasterizer: A Differentiable Renderer for Image-based 3D Reasoning," *IEE International Conference on computer vision*, pp. 7708- 7717, 2019.
- [14] J. Cowles, "Differentiable rendering," Medium , 19 08 2018. [Online]. Available: <https://medium.com/towards-data-science/differentiable-rendering-d00a4b0f14be>. [Accessed 20 07 2022].
- [15] T. Ngyuen-Phuoc, C. Li, S. Balaban and Y.-. L. Yang, "RenderNet: A deep convolutional network for differentiable rendering from 3d shapes," *Neural Information processing systems*, 2018.
- [16] H. Kato and T. Harada, "Slef-supervised learning of 3d objects from natural images," *Computer vision and pattern recognition* , 2019 .
- [17] H. Kato, Y. Ushiku and T. Harada, "Neural 3D Mesh Renderer," *Computer Vision and Pattern Recognition* , 2017.
- [18] Y. Zhang, W. Chen, H. ling, J. Gao, Y. Zhang, A. Torralba and S. Fidler, "IMAGE GANS MEET DIFFERENTIABLE RENDERING FOR INVERSE GRAPHICS AND INTERPRETABLE 3D NEURAL RENDERING," *Computer vision and pattern recognition*, 2020.
- [19] N. Zubic and P. Lio, "An effective loss function for generating 3D models from single 2D image without rendering," in *Artificial Intelligence Applications and Innovations*, 2021, pp. 309-322.
- [20] X. Xia, F. Y. Guan, Y. Cai and N. M. Thalmann, "Challenges and Advancements for AR Optical See-Through Near-Eye Displays: A Review," *Frontiers in Virtual Reality*, 2022.
- [21] X. Hu, Research gate, [Online]. Available: https://www.researchgate.net/figure/Main-components-of-a-near-eye-optical-see-through-headset_fig2_347698540. [Accessed 25 08 2022].
- [22] D. Ni, D. Cheng and Y. Wang, "Design of diffractive waveguide near eye display system with exit pupil expansion," *Display Technology* , 2021.
- [23] C. Yoo, K. Bang, M. Chae and B. Lee, "Extended-viewing-angle waveguide near-eye display with a polarization-dependent steering combiner," *Optical Letters*, vol. 45, no. 10, pp. 2870-2873, 2020.
- [24] JiashengXiao, JuanLiu, j. Han and Y. Wang, "Design of achromatic surface microstructure for near-eye display with diffractive waveguide," *Optics communication* , vol. 452, pp. 411-416, 2019 .
- [25] Y. W. C. X. W. S. a. G. J. Dewen Cheng, "Design of an ultra-thin near-eye display with geometrical waveguide and freeform optics," *Optics Express*, vol. 22, no. 17, pp. 20705-20719, 2014.

- [26] K. Sarayeddine and K. Mirza, "Key challenges to affordable see-through wearable displays: the missing link for mobile AR mass deployment," *Photonic Applications for Aerospace, Commercial, and Harsh Environments*, vol. 8720, 2013.
- [27] W. J. C. Bernard C. Kress, "Invited Paper: Towards the Ultimate Mixed Reality Experience: HoloLens Display Architecture Choices," *SID Symposium Digest of Technical Papers*, vol. 48, no. 1, 2017 .
- [28] K. Y. L.-T. W. Jianghao Xiong, "Holographic Optical Elements for Augmented Reality: Principles, Present Status, and Future Perspectives," *Advanced Photonics Research*, vol. 2, no. 1, 2020.
- [29] H.-J. K. S.-B. K. H. Z. B. L. Y.-M. J. S.-H. K. a. J.-H. P. Han-Ju Yeom, "3D holographic head mounted display using holographic optical elements with astigmatism aberration compensation," *Optics Express*, vol. 23, no. 25, pp. 32025-32034, 2015.
- [30] D. H. Close, "Holographic Optical Elements," *Optical engineering* , 1975.
- [31] A. Cameron, "The application of holographic optical waveguide technology to the Q-Sight family of helmet-mounted displays," *Head- and Helmet-Mounted Displays XIV: Design and Applications*, vol. 7326, 2009.
- [32] B. C. Kress and W. J. Cummings, "11-1: Invited Paper: Towards the Ultimate Mixed Reality Experience: HoloLens Display Architecture Choices," *SID Symposium Digest of Technical Papers*, vol. 48, no. 1, pp. 127-131, 2017.
- [33] Facebook Research, "Holographic optics for thin and lightweight virtual reality," Meta , 29 June 2020. [Online]. Available: <https://research.facebook.com/blog/2020/06/holographic-optics-for-thin-and-lightweight-virtual-reality/>. [Accessed 21 07 2022].
- [34] B. G, *Modern classical optics*, Oxford: Oxford University press, 2003.
- [35] A. L. D. R. K. K. K. L. D. a. F. H. Maimone, "Pinlight displays: wide field of view augmented reality eyeglasses using defocused point light sources," *ACM Transactions on Graphics*, vol. 33, no. 4, pp. 1-11, 2014.
- [36] K. AK, SIT, J. KAUTZ and D. LUEBKE, "Slim near-eye display using pinhole aperture arrays," *Applied Optics*, vol. 54, no. 11, pp. 3422-3427, 2014.
- [37] Wikipedia , "Coded aperture," Wikipedia , 13 09 2021. [Online]. Available: https://en.wikipedia.org/wiki/Coded_aperture. [Accessed 25 07 2022].
- [38] T. cannon and E. Fenimore, "Coded Apertures imaging: many holes make light work," *Optical Engineering* , vol. 19, no. 3, pp. 283-289, 1980 .

- [39] The Centre for X-Ray Optics, “Theory: Fresnel zone plate theory and equations,” The Centre for X-Ray Optics, 2014. [Online]. Available: <http://zoneplate.lbl.gov/theory>. [Accessed 25 07 2022].
- [40] R. Dicke, “Scatter-Hole Cameras for X-Rays and Gamma Rays,” *Astrophysical Journal*, vol. 153, p. L101, 1968.
- [41] “A television scanning scheme for a detector-noise-limited system,” *IEEE Transactions on Information Theory*, vol. 14, no. 3, 1968 .
- [42] “Light Field head-mounted display with correct focus cue using microstructure array,” *Chinese Optical Letters* , vol. 12, no. 6, 2014 .
- [43] W. song, Q. Cheng, P. Surman, Y. Liu, Y. Zheng, Z. Lin and Y. Wang, “Design of light-field near eye display using random pinholes,” *Optics Express*, vol. 27, no. 17, 2019.
- [44] H. Huang and H.Hua, “Systematic Characterization and optimisation of 3D light field displays,” *Optics Express*, vol. 25, no. 16, pp. 18508-18525, 2017.
- [45] J. smith, “Three applications of optimisation in computer graphics,” Carnegie Mellon University, school of computer science, Pittsburgh,, 2003.
- [46] T. Whitted, “An Improved Illumination Model for Shaded Display,” *Communications of the ACM*,, vol. 23, no. 6, pp. 342-349, 1980.
- [47] R. A. H. a. D. P. Greenberg, “A Testbed for Realistic,” *IEEE*, vol. 3, no. 10, pp. 10-20, 1983.
- [48] J. Arvo and D. Kirk, “A survey of ray tracing acceleration techniques,” in *An introduction to ray tracing* , 1990, pp. 204-205.
- [49] B. Macukow, “Neural Networks – State of Art, Brief History, Basic Models and Architecture,” *IFIP International Conference on Computer Information Systems and Industrial Management*, vol. 9842, pp. 3-14, 2016.
- [50] S. K. P. Nikhil R Pal, “A review on image segmentation techniques,” *Pattern Recognition* , vol. 26, no. 9, pp. 1277-1294, 1993.
- [51] M.Egmont-Petersen, D. Ridder and H.Handels, “Image processing with neural networks—a review,” *Pattern Recognition*, vol. 35, no. 10, 2002.
- [52] R. G. A. Adler, “A neural network image reconstruction technique for electrical impedance tomography,” *IEEE Trans. Med. Imaging*, vol. 13, no. 4, pp. pp 594-600, 1994.
- [53] G. Jiang and B. Kainz, “Deep radiance Caching: Convolutional autoencoders deeper in ray tracing,” *Computers and Graphics* , 2020.

- [54] H. Lee, C. Jo and K.-Y. Lee, "Implementation of Autoencoder based Neural Network for Realtime Ray Tracing," *International Journal of Hybrid Information Technology*, vol. 13, no. 1, pp. 1-6, 2020.
- [55] Nvidia, "Interactive Reconstruction of Monte Carlo Image Sequences using a," [Online]. Available: https://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf. [Accessed 14 august 2022].
- [56] Sky engine, "RAYTRACING ENGINE DEDICATED TO DEEP NEURAL NETWORKS TRAINING IN VIRTUAL REALITY," London, United Kingdom.
- [57] M. Weidmann, "Nvidia's AI Powered Ray Tracing: DLSS," 03 september 2020. [Online]. Available: <https://medium.com/@maxwiedmann/nvidias-ai-powered-ray-tracing-dlss-d0e59ab45232>. [Accessed 14 August 2022].
- [58] O. Aflak, "Ray Tracing From Scratch in Python," Medium, 26 July 2020. [Online]. Available: <https://medium.com/swlh/ray-tracing-from-scratch-in-python-41670e6a96f9>. [Accessed 17 June 2022].
- [59] P. Pendamkar, "Raytracing algorithm," EDUCBA, [Online]. Available: <https://www.educba.com/ray-tracing-algorithm/>. [Accessed 04 08 2022].
- [60] "What Is Gradient Descent In Machine Learning?," Tech Blog , 22 June 2018. [Online]. Available: <https://saugatbhattarai.com.np/what-is-gradient-descent-in-machine-learning/gradient-descent-2/>. [Accessed 14 August 2022].
- [61] S. Ruder, "An overview of gradient descent optimisation Algorithms," *ArXiv*, 2016.
- [62] L. Deng, "The mnist database of handwritten digit images for machine learning research.," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141-142, 2012.
- [63] L. Agapito, "Github," [Online]. Available: <https://github.com/LouAgapito/ImPro26/tree/main/images>. [Accessed 05 09 2022].
- [64] "Pixelart," [Online]. Available: <https://www.pixilart.com/draw>.
- [65] P. K. Das, "History and evolution of GPU architecture," in *Emerging research surrounding power consumption and performance issues in utility computing*, 2016, p. 27.
- [66] M. Patel, "When two trends fuse: Pytorch and recommender systems," O.REILLY, 07 12 2017 . [Online]. Available: <https://www.oreilly.com/content/when-two-trends-fuse-pytorch-and-recommender-systems/>. [Accessed 08 08 2022].
- [67] C. L. Laboratory, "Odak," GitHub, [Online]. Available: <https://github.com/kaanaksit/odak>. [Accessed 08 14 2022].
- [68] B. Ashworth, "What is Ray tracing? The Latest Gaming Buzzword Explained," Wired, 14 June 2019 . [Online]. Available: <https://www.wired.com/story/what-is-ray->

tracing/#:~:text=First%20conceptualized%20in%201969%2C%20ray,technology%20requires%20considerable%20computing%20power.. [Accessed 14 June 2022].

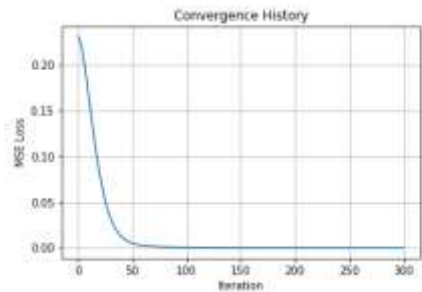
- [69] H. Kato, D. Beker, M. Morariu, T. Ando, T. Matsuoka, W. Kehl and A. Gaidon, “Differentiable Rendering: A Survey,” *Journal of Latex Class Files*, vol. 14, no. 8, 2015.
- [70] Synopsys, “What is Ray tracing,” [Online]. Available: <https://www.synopsys.com/glossary/what-is-ray-tracing.html>. [Accessed 05 07 2022].
- [71] N. V. Oosterwyck, Research Gate, [Online]. Available: https://www.researchgate.net/figure/Pinhole-camera-model-with-a-point-PX-Y-Z-according-to-the-camera-coordinate-system_fig2_326717734. [Accessed 2022 08 4].
- [72] B. Lamparter, H. Miller and J. Winckler, “The Ray-z- Buffer - An Approach for Ray Tracing Arbitrarily Large Scenes,” 1991.

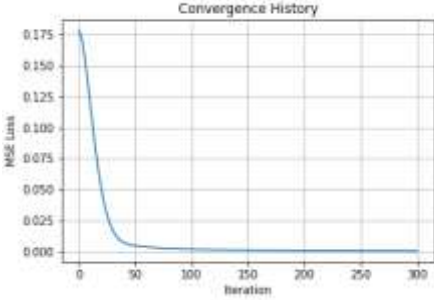
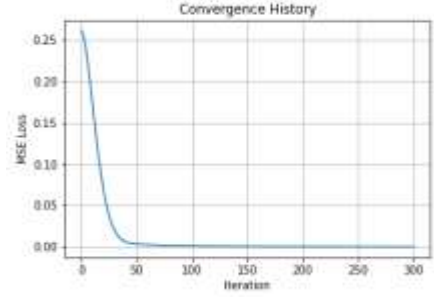
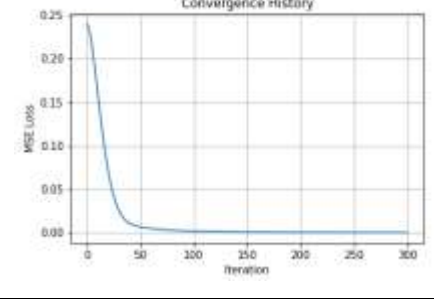
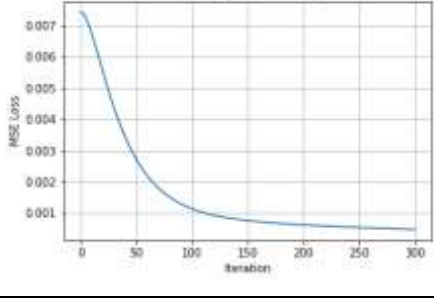
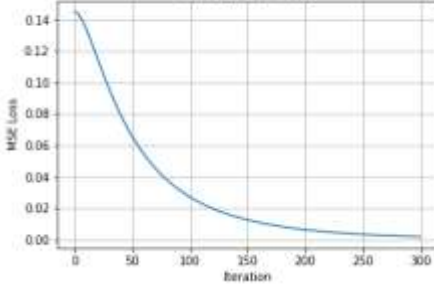
Chapter 8: Appendix

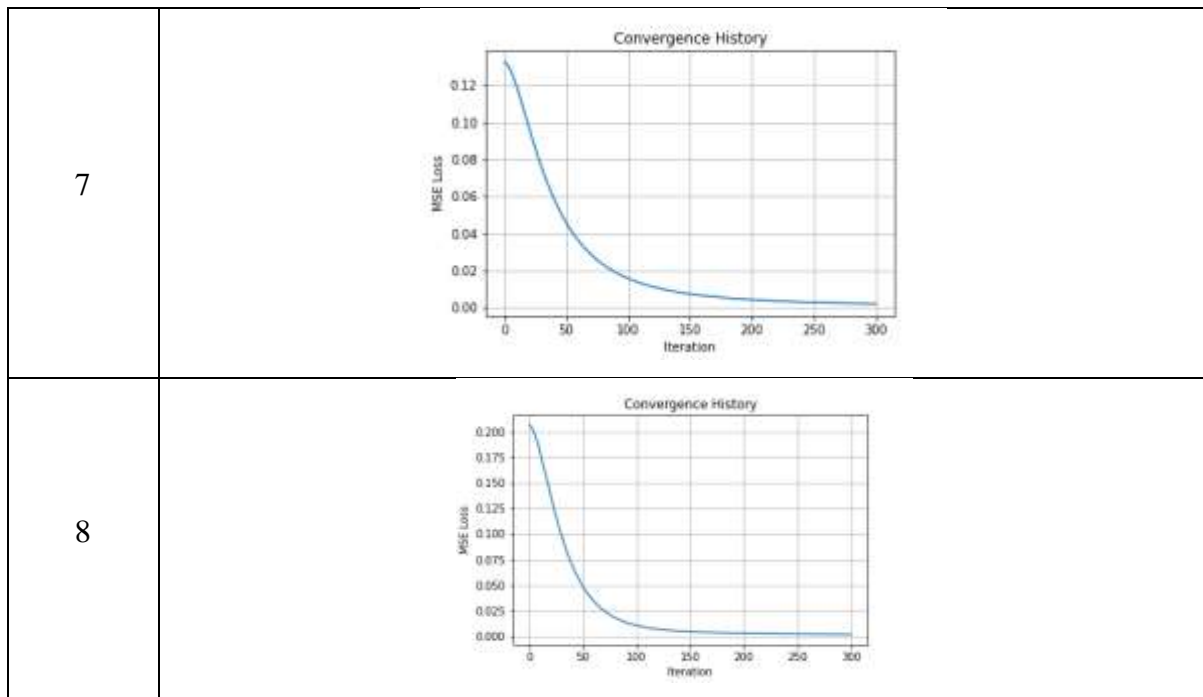
8.1. Terminology

- **AR:** Augmented reality
- **VR:** Virtual reality
- **Dynamic Random Access Memory (DRAM):** A type of semiconductor memory that is used for high memory capacity at low costs.
- **PSF:** Point spread function is the impulse response of an imaging system to a point source.
- **NEDs:** Near-eye displays
- **GPU's:** Graphical processing units
- **CPU:** Central processing unit
- **VGA:** Video Graphic Arrays
- **Hessian matrix:** A matrix contain the second order partial derivatives of a function
- **Second-order derivative:** The rate of change of a functions change
- **First-order derivative:** A measurement of the rate of change of a function
- **RGB:** Red, Green and Blue colour channels.
- **MNIST:** Modified National Institute of Standards and Technology
- **AI:** Artificial Intelligence

8.2. Convergence History

Test	Convergence History
1	 <p>The graph, titled 'Convergence History', plots 'MSE Loss' on the y-axis (ranging from 0.00 to 0.20) against 'Iteration' on the x-axis (ranging from 0 to 300). A blue line shows the loss starting at approximately 0.25 at iteration 0, decreasing rapidly to about 0.05 by iteration 25, and then continuing to decrease more slowly, reaching near zero by iteration 50. The loss remains stable at this low level through iteration 300.</p>

2	
3	
4	
5	
6	



8.3. Project Timeline

The project was scheduled to take 14 weeks in total, with time set aside for testing, analysis, and the possible creation of a 3D-printed model of the optimised aperture array. However, the design and implementation stage took longer than expected, thus plans to create a 3d model were scrapped. A draft of the initial of the entire project can be seen below:

Differentiable Neural Near-eye Display Design (PLAN)													
Write-Ups													
Coding													
Updates(If necessary)													
WK1	WK2	WK3	WK4	WK5	WK6	WK7	WK8	WK9	WK10	WK11	WK12	WK13	WK14
10/06/2022	17/06/2022	24/06/2022	01/07/2022	08/07/2022	15/07/2022	22/07/2022	29/07/2022	05/08/2022	12/08/2022	19/08/2022	26/08/2022	02/09/2022	09/09/2022
Introduction/Abstract													
	Literature review research and write up												
			System Design and Implementation										
			Write up of Design, Implementation, Testing and result										
					Designing Tests and system testing								
					Results and technique Improvements								
							Trouble shooting and System evaluation						
									Additional write up				
							Integrating system with prototype glasses						

Figure 11: Initial project Plan

8.4. Code

```

1. class aperture_array():
2.
3.     def __init__(self, device):
4.         self.device = device
5.         self.init_light_sources()
6.         self.init_aperture_array()
7.         self.init_detector()
8.
9.     # Transform image from PIL to normalised tensor
10.    def transform_image(self, image):
11.        """
12.        Resizes image, Transforms image to tensor \
13.        and normalises image
14.
15.        Parameters
16.        -----
17.        Image: Grayscale or Coloured PIL image
18.
19.        Returns
20.        -----
21.        normalised_image: Tensor of normalised and resized image
22.        """
23.
24.        #If grayscale then:
25.        if(len(np.array(image).shape)<3):
26.
27.            transform =transforms.Compose([
28.                transforms.Resize((28,28)),
29.                transforms.ToTensor(),
30.                transforms.Normalize((0.5), (0.5))])
31.
32.            im=transform(image)
33.            normalised_image = im.view(-1,1).to(self.device)
34.
35.
36.        #If it is a colour image then:
37.        elif int(np.array(image).shape[2]) == 3 or \
38.            int(np.array(image).shape[2]) == 4:
39.
40.            transform = transforms.Compose([
41.                transforms.ToTensor(),
42.                transforms.Resize((28,28))])
43.
44.            pixel = np.asarray(image)
45.            # Convert from integers to floats
46.            pixel = pixel.astype('float32')
47.            # Normalize to the range 0-1
48.            pixel /= 255.0
49.            im = torch.moveaxis(transform(pixel[:,:,:3]), 0,2)

```

```

50.     normalised_image = im.view(-1,3).to(self.device)
51.
52.     return normalised_image
53.
54.
55. # Initialising light sources
56. def init_light_sources(self, dimensions=[0.01, 0.015], \
57.                        pixel_count=[28,28], Z=0.):
58.
59.     """
60.     Defines the light source's locations
61.
62.     """
63.     x = torch.linspace(-dimensions[0]/2., \
64.                        dimensions[0]/2., pixel_count[0])
65.
66.     y = torch.linspace(-dimensions[1]/2., \
67.                        dimensions[1]/2., pixel_count[1])
68.     X, Y = torch.meshgrid(x, y, indexing='ij')
69.     self.light_source_locations = torch.zeros(\
70.     X.shape[0], X.shape[1], 3).to(self.device)
71.
72.     self.light_source_locations[:, :, 0] = X
73.     self.light_source_locations[:, :, 1] = Y
74.     self.light_source_locations[:, :, 2] = Z
75.
76. # Defining Array
77. def init_aperture_array(self, dimensions=[0.01, 0.015], \
78.                        pixel_count=[28,28], Z=0.01):
79.
80.     """
81.     Defines the aperture array and its locations
82.
83.     Returns
84.     -----
85.     self.aperture_array: A 28 x 28 Tensor
86.     of random values
87.
88.     """
89.     x = torch.linspace(-dimensions[0]/2., \
90.                        dimensions[0]/2., pixel_count[0])
91.     y = torch.linspace(-dimensions[1]/2., \
92.                        dimensions[1]/2., pixel_count[1])
93.     X, Y = torch.meshgrid(x, y, indexing='ij')
94.     self.aperture_array_locations = torch.zeros(\
95.     X.shape[0], X.shape[1], 3).to(self.device)
96.     self.aperture_array_locations[:, :, 0] = X
97.     self.aperture_array_locations[:, :, 1] = Y
98.     self.aperture_array_locations[:, :, 2] = Z

```

```

99.     self.aperture_array = torch.rand(X.shape[0], \
100.                                     X.shape[1],requires_grad=True, device=self.device)
101.
102.         return self.aperture_array
103.
104.         # Defining plane
105.     def init_detector(self, dimensions=[0.01, 0.015],\
106.                      size=[28,28], Z=0.01):
107.         """
108.         Defines planar surface
109.
110.         """
111.
112.         point = torch.tensor([0., 0., Z]).to(self.device)
113.         self.detector_surface = \
114.             odak.learn.raytracing.define_plane(point)
115.
116.         # Forming image on detector surface
117.     def intersection_points_to_image(self, points, \
118.                                     amplitudes, threshold, norm_image):
119.         """
120.         Displays image generated from
121.         the intersection points from ray tracing
122.
123.         Parameters
124.         -----
125.
126.         points: Tensor containing intersection point of \
127.         the ray on the planar surface
128.
129.         amplitudes: Tensor of the aperture array
130.
131.         threshold: 1x1 Tensor of the distance between \
132.         pixels on the detector surface
133.
134.         norm_image: Tensor of the normalised \
135.         and resized image
136.
137.         Returns
138.         -----
139.
140.         detector_image: Tensor containing\
141.         the image formed from \
142.         intersection points on the image plane
143.         """
144.         Points_data= norm_image * amplitudes
145.         detector = torch.zeros_like(\
146.             norm_image).to(self.device)
147.

```



```

148.         array_locations = \
149.             self.aperture_array_locations.view(-1, 3)
150.
151.         for idx, point in enumerate(points):
152.
153.             dist_btwn_array_npoint = torch.sqrt(\
154.                 torch.sum((array_locations-point)**2, dim=1))
155.
156.             min_dist_idx = torch.argmin(\
157.                 dist_btwn_array_npoint)
158.             min_dist = torch.min(dist_btwn_array_npoint)
159.
160.             if min_dist < (math.sqrt(2) * threshold):
161.                 detector[min_dist_idx] = Points_data[idx]
162.
163.         detector += detector
164.
165.         # resizing depending on if image \
166.         is colour or grayscale
167.
168.         if int(norm_image.size()[1]) == 3:
169.             detector_image = \
170.                 detector.view(28,28,3).to(self.device)
171.         else:
172.             detector_image = \
173.                 detector.view(28,28).to(self.device)
174.
175.         return detector_image
176.
177.     # Forward to do ray tracing \
178.     return intersection points and get image
179.     def forward(self, array, image):
180.
181.         """
182.         Calculates the intersection points
183.         of the ray with the planar surface
184.         and returns the resultant image
185.
186.         Parameters
187.         -----
188.         array: Tensor containing the aperture array
189.
190.         image: PIL image
191.
192.         Returns
193.         -----
194.         detector_image: Tensor of
195.         resultant image from ray tracing
196.

```

```

197.         intersections: List containing \
198.         the intersection points of the ray with the plane
199.
200.         """
201.
202.         light_source_locations = \
203.         self.light_source_locations.view(-1, 3)
204.
205.         aperture_array_locations = \
206.         self.aperture_array_locations.view(-1, 3)
207.
208.         self.pixel_pitch = torch.sqrt(\
209.         torch.sum((aperture_array_locations[1] - \
210.         aperture_array_locations[0])**2, \
211.         dim = 0)).to(self.device)
212.
213.         aperture_array = array.view(-1, 1).to(self.device)
214.
215.         self.image = image
216.         self.norm_img = self.transform_image(self.image)
217.         intersections = []
218.
219.         for light_source_location in light_source_locations:
220.             rays_from_light_source = \
221.             odak.learn.raytracing.\
222.             create_ray_from_two_points\
223.             light_source_location, \
224.             aperture_array_locations)
225.
226.             # calculating intersection points
227.             intersection_normals_w_detector, _ = \
228.             odak.learn.raytracing.intersect_w_surface\
229.             (rays_from_light_source, self.detector_surface)
230.
231.             intersection_points_w_detector = \
232.             intersection_normals_w_detector[:, 0]
233.             intersections.append(\
234.             intersection_points_w_detector)
235.
236.             # Getting the image seen on the image plane
237.             image = list(map(self.intersection_points_to_image, \
238.             intersections, [aperture_array], \
239.             self.pixel_pitch), [self.norm_img]))
240.
241.             detector_image = image[0]
242.             return detector_image, intersections
243.
244.     def compute_loss(self, output, target):
245.         """

```

```

246.         Calculates the mean square error between two values
247.
248.     Parameters
249.     -----
250.
251.     output: Tensor of image from ray tracing
252.
253.     target: Tensor of the normalised input image
254.
255.     Returns
256.     -----
257.
258.     loss: Tensor of the mean squared loss value
259.
260.     """
261.     loss = torch.nn.MSELoss()(output, target)
262.
263.     return loss
264.
265. def optimize(self):
266.
267.     """
268.     Optimises the aperture array with
269.     stochastic gradient descent
270.
271.     Parameters
272.     -----
273.
274.     intersection_points: List containing \
275.     the intersection\
276.     points of the ray with the plane
277.
278.     Returns
279.     -----
280.
281.     model: Tensor of the optimised array
282.
283.     history: List containing loss value
284.     in every iteration
285.     """
286.     if int( self.norm_img.size()[1]) == 3:
287.         target = self.norm_img.view(28,28,3)
288.     else:
289.         target = self.norm_img.view(28,28)
290.
291.     array = self.aperture_array.view(-1,1)
292.     array_loc = self.aperture_array_locations
293.     model = [nn.Parameter(array)]
294.     points= intersection_points
295.     threshold= self.pixel_pitch

```

```
294.
295.     optimiser= torch.optim.SGD(\
296.     model,lr = 0.8,momentum=0.9)
297.
298.     epochs= 300
299.     history =[]
300.
301.     for epoch in range(0, epochs+1):
302.         optimiser.zero_grad()
303.
304.         output, _=self.forward(model[0],self.image)
305.         loss = self.compute_loss(output,target)
306.
307.         loss.backward()
308.         optimiser.step()
309.
310.         print('Epoch: ', epoch, ' ,Loss: ', loss)
311.         history.append(loss)
312.
313.     return model, history
```