Optical Depth Estimation Using Q-Nets

Yannik Nelson

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

Real time ray-tracing has become an increasingly popular technology in recent years, due in large part to the games industry's push for realism. Volumetric rendering is a particularly difficult type of rendering that, in real-time applications, is often substituted for approximate solutions. Volumetric rendering requires calculating the optical depth of a ray through a volume in order to determine how much light exits the volume along that ray [27]. The optical depth of a ray can be described as the total density encountered by that ray through a volume.

In this dissertation, we have developed a method to use neural networks to speed up optical depth estimation, which we call the Q-Net rendering method [26]. The Q-Net rendering method represents the volumetric data using a neural network and then integrates the result space of that network to find the optical depth of specified rays. We discuss the background knowledge required to understand volumetric rendering, neural networks, and the integration of neural networks. We then describe and evaluate the method developed, finding that our method uses less memory and less time than our reference technique, ray-marching.

Acknowledgements

- Kartic Subr: Supervisor
- Rachel Noach: Proof reading
- Somhairle Macleòid: Proof reading
- Ariane Branigan: Proof reading
- Ponrawee Prasertsom: Proof reading
- Martin Keith: Proof reading
- Claire Stoneham: Proof reading

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Volumetric rendering is used to render non-solid objects such as clouds or smoke. This has applications in many computer graphics scenarios from animated movies and video games rendering mist, clouds or explosions. In order to render these objects, we need to be able to calculate the proportion of incoming light that exits the object in a specified direction. This can be calculated using the optical depth of rays through the volume.

The optical depth of a ray is the total density encountered by that ray through a volume [27]. If we consider the density distribution within the volume as a function, we can describe the optical depth of a ray as the integral of the volume density along that ray. When the volume is homogeneous (the density is constant throughout the volume), the optical depth is the density of the volume $\times$ the distance the ray travels through the volume [27]. However, when the volume is non-homogeneous, this integral is harder to calculate.

The majority of optical depth estimation techniques use numerical methods such as ray-marching or Monte Carlo integration. The numerical methods take samples along the ray being integrated. The run-time of numerical methods increases with the number of samples taken and the range over which those samples were taken, with higher samples rates producing more accurate results. As the optical depth of each ray is required, improving the run-time of optical depth estimation would result in a dramatic overall speed-up.

There has been recent research into optimising the integration of a neural network's result space [26, 18, 28]. This optimisation can be summarised as a reformulation of the integral formula [28, 18] which allows part of the formula to be evaluated as a neural network. The research, [26], applied this method to estimating optical depth along rays in two dimensions.

We have extended this to estimating the optical depth along rays in three dimensions. This is has useful applications for industry rendering technologies including: reducing

the data size requirements by holding only the neural network weights, as well as speeding up rendering by making optical depth approximation constant time.

1.2 Research Objectives

The goal of this dissertation is to act as a proof of concept for volume rendering utilising Q-Nets as well as to show that the use of Q-Nets in volume rendering has merit in terms of speed up and memory usage.

We shall evaluate the run time and accuracy, comparing it to the run time of a well established alternative method.

1.2.1 Research Questions

1. Is Optical depth estimation possible in three dimensions using Q-Nets?
2. Is optical depth estimation using Q-Nets faster than current methods?
3. What are the characteristics of the Q-Net optical depth estimation run-time?
4. What scenarios produce high inaccuracy?

Chapter 2

Background

2.1 Ray-Tracing

2.1.1 Radiometry vs Photometry

According to [4], there are two general ways of measuring the forms of radiation we perceive as light: **radiometry** and **photometry** . Radiometry is concerned with radiation of all wavelengths . Photometry is concerned with only the radiation and wavelengths visible to the human eye.

For both methods of measurement, a 'reading' of radiation is weighted (multiplied) by the result of a function $v(\lambda)$ [4]. In radiometry $v(\lambda)$ maps the wavelength of the radiation measured to the energy a photon at this wavelength has [4]. In photometry $v(\lambda)$ is known as the **luminous efficiency** and maps the wavelength to the eye's sensitivity to that wavelength [4]. The luminous efficiency function can be split into three parts, $v_R(\lambda)$, $v_G(\lambda)$ and $v_B(\lambda)$, each corresponding to the sensitivity of the Red, Green and Blue cones (within the eye) to wave length λ [4].

We are interested in the light energy in the scene across the visual spectrum. As such we are only interested in radiometry and radiance.

2.1.2 Radiance, Irradiance and Flux

As discussed in [4], the radiometric measures we are concerned with are:

- **Radiant energy**, measured in Joules (J), is the energy found in electromagnetic waves.
- **Radiant flux**, measured in watts (W), is the change in radiant energy per second. As an example, the sun emits $3.846 \times 10^{26} W$.
- **Radiance**, measured in watts per **steradian** ($W.\text{sr}^{-1}.\text{m}^{-2}$), is the radiant flux flowing in a given direction. This metric represents what an eye or camera sees.

  Note: a steradian is the unit of solid angles, a term used to define a set of directions. A complete sphere is $4\pi sr$ and $2\pi sr$ is a hemisphere [4].

- **Irradiance**, measured in watts per square meter ($W.\mathrm{m}^{-2}$), is the total radiance received at a point from all directions.

The equivalent photometric measures are **luminous energy**, **luminous flux**, **luminance**, and **illuminance**, respectively[4].

The use of these terms in literature is often inconsistent and as such, we shall be using the terms consistent with the sources used.

2.1.3 The Rendering Equation

There are many rendering algorithms, each providing unique trade-offs between accuracy and speed. All of these rendering algorithms approximate a general equation [17]:

$$I(x,x') = g(x,x')\left[\varepsilon(x,x') + \int_S p(x,x',x'')I(x',x'')\mathrm{d}x''\right] \tag{2.1}$$

In this equation:

$I(x,x')$	is the luminance from point x' that reaches point x
$g(x,x')$	is the geometry term
$\varepsilon(x,x')$	is the illuminance from x' to x
$p(x,x',x'')$	is the luminance scattered from x'' to x by a patch of surface at x'
S	is the union of all surfaces

The geometry term encodes occlusion of points on a surface by other surfaces. If the points are occluded then the term is 0, if they are visible to each other, then the term is $\frac{1}{r}$ where r is the distance between x and x', this term does not take material into account[17].

2.1.4 Geometric Optics

Geometric optics treats light rays as geometric lines along which energy flows. In geometric optics, rays are defined as vectors that are normal to the wave-front of the light they are describing [13]. Geometric optics is based on 5 basic observations [13]:

1. Light travels in a straight line in homogeneous media
2. The law of reflection
3. The law of refraction
4. Light beams propagate without disturbing each other
5. Light beams from 'natural' sources of light generally do not show interference.

The cases where the first and last observation are violated can only be fully observed through accurate experimentation [13].

2.1.5 Ray-Tracing

Ray-tracing is one of the algorithms that approximates the rendering equation, mentioned in section 2.1.3. The ray-tracing algorithm takes advantage of the vector format of light rays described by geometric optics [24]. A ray is generated for each pixel of a render, an intersection test is then performed on all the objects in the scene. This finds if the ray of interest would intersect the object being tested. The intersection closest to the source of the ray is then used to calculate the colour of the pixel associated with that ray.

The contribution to a pixel's colour from an intersection is the sum of lighting components at the point of intersection. These components, as described in [24], are:

- The ambient light
- The diffuse light
- The specular light
- The self-luminosity
- The reflected light
- The refracted light

Each of these quantities are calculated using the vector of the ray, the normal at the intersection point and the vectors from the intersection point to the light sources. The contributions due reflected light and refracted light are calculated by finding the reflected and refracted ray directions and then calculating the contribution of those rays recursively [22].

Due to the potentially infinite recursion in this process, a limit on recursive depth is used.

The form of ray-tracing as described is limited to sharp shadows, reflections and refraction, due to the precise nature of geometric calculations. A method known as distributed ray tracing is described in [7]: this is a form of anti-aliasing that uses extra ray samples to produce 'fuzzy' phenomena such as soft shadows and lens focus.

The requirement of extra ray samples in distributed ray-tracing increases the run-time of the ray-tracer. For example, if a ray-tracer using distributed ray-tracing allows a recursion depth of three, 100 samples per shadow cast and 100 samples from the camera lens per pixel, then the number of rays for a single pixel will be:

$$\text{camera samples} \times 2^{\text{recursion depth}} - 1 \times (\text{shadow cast samples} + 1) = 70700$$

This assumes the involvement of both reflection and refraction at each recursion step and only one light source, producing two recursive calls per intersection. This results in $2^{\text{recursion depth}} - 1$ rays per camera sample, each of which will also require 100 rays per light source to calculate the lighting.

As such, methods to approximate these processes with fewer samples are often used and are an area of significant research interest.

2.2 Volumetric Rendering

The volumes of solid objects are often ignored when being rendered, as only the surface can be seen and therefore needs to be considered. Volumetric rendering is concerned with the cases where this is not true. For example clouds, where the distribution of water droplets in the cloud affect how light travels through it.

In volumetric rendering, we are concerned with how light travels through a volume. To calculate this we focus on three main phenomena—**absorption**, **emission** and **scattering** [27]:

- Absorption occurs when a photon hits a particle in the volume and is absorbed by that particle, stopping the photon's transport through the volume.
- Emission occurs in volumes when particles emit light.
- Scattering occurs when light hits a particle in the volume and is reflected off into a new direction. We are only concerned with two cases of scattering:
  - **Out-scattering**, where light traveling in the direction of interest is scattered and begins traveling in a direction that will not contribute the luminance of interest.
  - **In-scattering**, where light travelling in other directions scatters into the direction of interest.

The difference between the luminance entering a volume and the luminance exiting a volume is $\Delta L = \text{emission} + \text{scattering}_{\text{in}} - \text{scattering}_{\text{out}} - \text{absorption}$ [27]. For the purposes of this dissertation, only absorption and scattering shall be considered, treating emission as 0. This gives us the equation $\Delta L = \text{scattering}_{\text{in}} - \text{scattering}_{\text{out}} - \text{absorption}$.

The following sections use a simplified mathematical model to finding the luminance L received by a viewer looking in direction $\overrightarrow{\omega}$ through a cylinder dV, with length ds and cross section $d\sigma$. We will consider ΔL to be the change in luminance across dV (a section of volume) at point $\boldsymbol{P}$, the center of the cylinder, with direction $\overrightarrow{\omega}$.

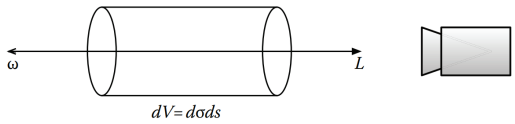


Figure 2.1: Differential volume along a view ray [27]

2.2.1 Absorption

The fraction of light a medium absorbs over a distance ds is known as the **absorption coefficient** or the **absorption cross section**, denoted σ_a , and is a reciprocal distance (m^{-1}) [27]. The inverse of σ_a is the mean free path or the average distance light will travel through the medium before interaction [27].

We are interested in finding the exiting luminance L_o, given an incoming luminance L_i and the absorption coefficient [27]:

$$L_o = L_i(\boldsymbol{p}, \overrightarrow{\omega}) + \mathrm{d}L_a \tag{2.2}$$

Where:

$$\mathrm{d}L_a = -\sigma_a L_i(\boldsymbol{p}, \overrightarrow{\omega})\mathrm{d}s \tag{2.3}$$

2.2.2 Scattering

Similarly to absorption, we describe the likelihood of a scattering event with a single coefficient known as the **scattering coefficient** usually denoted σ_s and again is a reciprocal distance (m^{-1}) [27].

We are interested in finding the exiting luminance L_o, given an incoming luminance L_i and the scattering coefficient:

$$L_o = L_i(\boldsymbol{p}, \overrightarrow{\omega}) + \mathrm{d}L_{out}(\boldsymbol{p}, \overrightarrow{\omega}) + \mathrm{d}L_{in}(\boldsymbol{p}, \overrightarrow{\omega})[27] \tag{2.4}$$

For out-scattering, as far as the observer is concerned, light disappearing due to scattering is the same as light disappearing due to absorption. As such we can use a similar equation to absorption:

$$\mathrm{d}L_{out}(\boldsymbol{p}, \omega) = -\sigma_s L_i(\boldsymbol{p}, \overrightarrow{\omega})\mathrm{d}s[27] \tag{2.5}$$

We can consider this in conjunction with absorption: out-scattering and absorption are known as **extinction** or **attenuation** when considered together, denoted σ_e [27].

For in-scattering, we need to consider the light sources in the scene; we represent the incoming radiance from a light source by $S(\boldsymbol{p}, \overrightarrow{\omega})$. We can find the amount of light from the light sources that has been scattered in the same way as out-scattering, but we now also need to know how much of the scattered light has been scattered in the direction of the observer. We can find this using the phase function $p(\overrightarrow{\omega}, \overrightarrow{\omega}')$ which tells us how much of the light traveling in direction $\overrightarrow{\omega}'$, is reflected in the direction $\overrightarrow{\omega}$. This gives us the equation:

$$\mathrm{d}L_{out}(\boldsymbol{p}, \omega) = -\sigma_s p(\overrightarrow{\omega}, \overrightarrow{\omega}')S(\boldsymbol{p}, \overrightarrow{\omega}')\mathrm{d}s[27] \tag{2.6}$$

2.2.3 Optical Depth

As described in [27], optical depth is a measure of transparency related to the distance light has to travel through a volume; the thicker the volume, the less light can penetrate. In the previous sections, we were only concerned with a homogeneous cylinder of volume of limited length; the equations need to be extended for entire rays in order to consider non-homogeneous cylinders.

It is possible to define the fraction of light that reaches a point distance s into a volume by integrating the equations for absorption and out-scatter [27]:

$$\begin{aligned} \mathrm{d}L(x) &= -\sigma L(x)\mathrm{d}x \\ \frac{\mathrm{d}l(x)}{L(x)\mathrm{d}x} &= -\sigma \\ \int_0^s \frac{\mathrm{d}l(x)}{L(x)\mathrm{d}x}\,\mathrm{d}x &= \int_0^s -\sigma\,\mathrm{d}x \\ \Big[ln(L(x))\Big]_0^s &= \Big[-\sigma x\Big]_0^s \\ ln(L(s)) - ln(L(0)) &= -\sigma s \\ e^{ln(L(s))-ln(L(0))} &= e^{-\sigma s} \\ \frac{L(s)}{L(0)} &= e^{-\sigma s} \end{aligned} \tag{2.7}$$

The relationship between $L(s)$ and $L(0)$ is known as **transmittance**. Equation 2.7 is known as Beer's law and can be written as $T = e^{-\tau}$, where T is the transmittance and τ is the optical depth [27]. If we assume the volume is non-homogeneous then $\tau = \int_0^d \sigma_e(\boldsymbol{p})\,\mathrm{d}s$ (where $\sigma_e(\boldsymbol{p})$ is the extinction at point $\boldsymbol{p}$ in the volume). In the case where the volume is homogeneous, then $\sigma_e(\boldsymbol{p}) = \sigma_e$ for all points and so $\tau = \sigma_e s$ [27].

It should be noted that σ_a and σ_s (and therefore σ_e) can be wavelength dependent [27].

From this, we can see that measuring the optical depth along a ray can tell us how much light is transmitted along that ray. This can in turn be used to inform the rendering of the pixel that ray originates from. For example, an adaptation of Beer's Law is used to render clouds in [25] .

Remember the optical depth τ is $\int_0^d \sigma_e(\boldsymbol{p})\,ds$, and that $\sigma_e(\boldsymbol{p})$ is a function on the location within the volume. Unfortunately, the format of volumetric data does not often lend itself to analytical integration, instead requiring a sampling method such as ray marching. Again, this is discussed in [25].

2.3 Neural Networks

From this point onward, vectors shall be denoted as bold lowercase characters (e.g. $\boldsymbol{x}$) and matrices shall be denoted as bold uppercase characters (e.g. $\boldsymbol{W}$) and superscript characters shall be used to denote the elements of a vector or matrix (e.g. $\boldsymbol{x}^i$ is the i_{th} element of $\boldsymbol{x}$ and $\boldsymbol{W}^i$ is the i_{th} row of $\boldsymbol{W}$).

2.3.1 Neurons

A neural networks consist of a network of linear units called neurons. Neurons can take any number of inputs, n, and produce one output, y. Each of the inputs x_i is multiplied by a weight w_i and then summed with a bias w_0, $\sum_{i=1}^n x_i w_i + w_0$. The output or activation of the neuron is the neuron's activation function $a(x)$ with this sum passed

in. Often the activation function is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. For ease of notation and implementation, this process is often presented in vector notation:

$$\boldsymbol{x} = (x_1, x_2, ..., x_n) \quad \boldsymbol{w} = (w_1, w_2, ..., w_n)$$

$$y = a\left(\boldsymbol{x}^T \boldsymbol{w} + w_0\right) = a\left(\sum_{i=1}^{n} x_i w_i + w_0\right) \text{ [23]} \tag{2.8}$$

On their own, each neuron simply performs logistic regression, but once in a network the results become much more sophisticated.

2.3.2 Networks

2.3.2.1 Fundamental Maths

A neural network consists of 'layers' of neurons. All the neurons in these layers have the same inputs and their own outputs. The inputs to each layer are the outputs of the previous layer, with the first layer's inputs being the data and the last layer's outputs being the output of the neural network [23].

Instead of calculating the outcome of each neuron in a layer one at a time, collecting them, and then passing them on; we can convert the operation into a matrix multiplication [23].

First we construct a matrix for the weights of each layer:

$$\boldsymbol{W}_i = \begin{pmatrix} w_{1,1} & w_{2,1} & \dots & w_{n,1} \\ w_{1,2} & w_{2,2} & \dots & w_{n,2} \\ \dots & & & \\ w_{1,m} & w_{2,m} & \dots & w_{n,m} \end{pmatrix}$$

Where $\boldsymbol{W}_i$ is the matrix corresponding to to connection from layer i (with $i = 0$ being the inputs), of size n, to layer $i+1$, of size m, and $w_{j,k} = \boldsymbol{W}^{j,k}$ is the weight from neuron j in layer i, to neuron k in layer $i+1$.

We then construct a vector for the inputs to layer i:

$$\boldsymbol{i}_i = \begin{pmatrix} i_1 \\ i_2 \\ \dots \\ i_n \end{pmatrix}$$

Where each $i_j = \boldsymbol{i}^j$ is the input to neuron j of layer i.

Finally, we construct the bias vector for the layer i:

$$\boldsymbol{w}_i = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_m \end{pmatrix}$$

Where each $w_j = \boldsymbol{w}^j$ is the bias for neuron j of layer i.

The multiplication of these two matrices plus the bias vector will then produce the weighted sum for each neuron in layer i, $\boldsymbol{W}_i\boldsymbol{i}_i + \boldsymbol{w}_i = \boldsymbol{O}_i$. To then create the inputs for the next layer, we apply the neurons' activation function to each element of $\boldsymbol{O}_i$, $a(\boldsymbol{O}_i) = \boldsymbol{i}_{i+1}$ [23].

It has been shown that neural networks with sinusoidal activation functions are general approximators, meaning they are capable of matching any function within an arbitrary error [8, 15].

2.3.2.2 Backpropagation

To train a neural network, we want to consider the relation of weights to errors in order to minimise the errors. For this, we want the derivative of the error with respect to the weights of the network, and we need a way to use this to calculate new weights $\Delta w_i = -\gamma\frac{\delta E}{\delta w_i}$ [23].

This process can be achieved with **backpropagation**. In order to perform backpropagation the derivative of the activation function a will be required, a'. In the case of the sigmoid function σ, the derivative is $\sigma' = \sigma(1-\sigma)$ [20]

When the activation functions are calculated for each layer i (excluding the input 'layer'), the derivative of those functions should be calculated for the same input value and stored in the diagonal entries of a diagonal matrix $\boldsymbol{D}_i$ [23].

Once the network has evaluated an item of test data, the final error can be calculated and placed in a vector $\boldsymbol{e}$ with each element of $\boldsymbol{e}$ being the error from the corresponding output node [23].

Assuming the network has $l+1$ layers, with the input layer being layer 0 and the output layer being layer l: the backpropagated error any layer $\delta^{(i)}$ is $\boldsymbol{D}_i\boldsymbol{W}_i\delta^{(i+1)}$ [23]. For the output layer this is simply $\delta^{(l)} = \boldsymbol{D}_l\boldsymbol{e}$ [23, 20]. The updates for the weights can then be calculated using $\Delta\boldsymbol{W}_i^T = -\gamma\delta^{(i+1)}\boldsymbol{i}_i$ [23]. Where γ is a small float value that regulates 'learning speed' called the **learning rate**.

To aid in understanding the variables, figure 3.5 holds a labelled illustration of such a network.

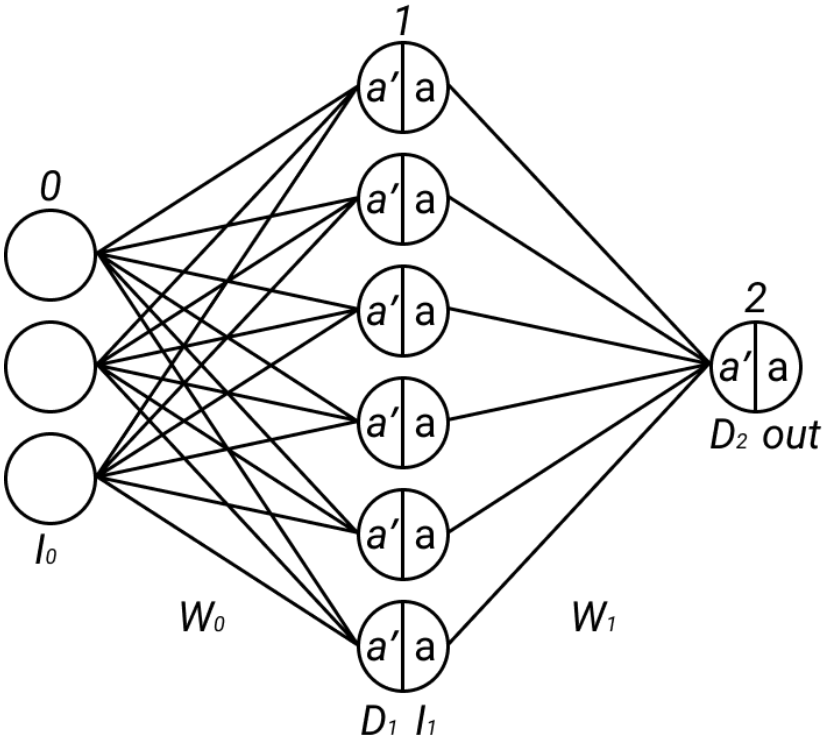


Figure 2.2: Simple neural-network structure

2.4 Q-Net

2.4.1 Integration Formula

According to [26], Q-Nets allow for the integration of the functions represented by a neural network of the form:

$$f_w(\boldsymbol{x}) = \boldsymbol{w}_2\sigma(\boldsymbol{W}_1\boldsymbol{x} + b_1) + b_2 \tag{2.9}$$

This is a neural network with d inputs, a sigmoid activated hidden layer with k neurons, and a linear output layer. As the inputs to this neural network will be normalised (between -1 and 1), we can define a hyper rectangular space $\mathcal{D} \equiv [-1,1]^d$ such that all $\boldsymbol{x} \in \mathcal{D}$ [26].

Q-Nets integrate such neural networks using the fact that the integral of a sum of shifted and scaled sigmoids is a weighted sum of the integrals of the sigmoids [18, 28]. In terms of the described network, the hidden layer is a series of shifted sigmoids, and the linear output layer scales and sums them [26]. As such the integral of the network can be found using this fact.

The formula for the integral of f_w, found in [26], is:

$$\mu_{d,k}(w) = \boldsymbol{w}_2\boldsymbol{v} + 2^d b_2 \tag{2.10}$$

Here, $\boldsymbol{v}$ is a column vector where:

$$\boldsymbol{v}^i = 2^d + \frac{1}{\tilde{w}_1^i} \sum_{m=1}^{2^d} a_m \mathrm{Li}_d(-e^{\boldsymbol{S}^m \boldsymbol{W}_1^{i\,T} - b_1^i}) \tag{2.11}$$

Here:

- $\mathrm{Li_d}$ is the polylogarithm function of order d.
- $\boldsymbol{S}$'s rows represent the 2^d vertices of the hyper rectangular domain the function's input space sits in $\mathcal{D}$. $\boldsymbol{S}^m$ is the m^{th} row representing the m^{th} vertex. All elements of $\boldsymbol{S}$ are either 1 or -1.
- a_m is the contribution at each of the hyper rectangular domain $\mathcal{D}'s$ vertexes. $a_m = \prod_{j=0}^{d} \boldsymbol{S}^{m,j}$, in other words, a_m is 1 if the number of -1s in $\boldsymbol{S}^m$ is even, -1 otherwise.
- $\tilde{w}_1^i \equiv \prod_{j=1}^{k} \boldsymbol{W}_1^{i,j}$ is the product of the elements of the i^{th} row of $\boldsymbol{W}_1$

2.4.2 Q-Nets

The Q-Nets, from [26], use a re-representation of the formula for $\boldsymbol{v}^i$ from the previous section:

$$\boldsymbol{v}^i = 2^d + \frac{q_d(y_i)}{\tilde{w}_1^i} \tag{2.12}$$

$$q_d(y_i) \equiv \boldsymbol{w}_3 \sigma_q\big(\begin{bmatrix}\boldsymbol{S} & -\mathbf{1}_{2^d}\end{bmatrix}\boldsymbol{y}_i\big)\boldsymbol{y}_i \equiv \begin{bmatrix}\boldsymbol{W}_1^i & \boldsymbol{b}_1^i\end{bmatrix}^T \tag{2.13}$$

Here:

- $\sigma_d(x) \equiv \mathrm{Li}_d(-exp(x))$
- $\mathbf{1}_{2^d}$ is a column of 2^d ones

This representation of $q_d(y_i)$ is of a similar form to that of a neural network and can be though of as such [26]. This neural network has: one hidden layer containing 2^d neurons; using the activation function σ_q; no bias'; input weights of $\begin{bmatrix}\boldsymbol{S} & -\mathbf{1}_{2^d}\end{bmatrix}$; output weights are $\boldsymbol{w}_3$; inputs are $\boldsymbol{y}_i$s; and the outputs are $q_d(\boldsymbol{y}_i)$s. Note that $\boldsymbol{w}_3$ is a row vector who's m^{th} element is a_m [26].

In practice, this can be vectorised for efficient computation making the inputs $\begin{bmatrix}\boldsymbol{W}_1 & \boldsymbol{b}_1\end{bmatrix}^T$ and $\boldsymbol{w}_3$ becomes $\boldsymbol{W}_3$ whose rows are all $\boldsymbol{w}_3$ [26].

Representing the equation in this form allows the $q_d(y_i)$ function to be evaluated like a neural network, taking advantage of the same hardware speedups and evaluating the qnet at all vertices of the hyper rectangular domain in parallel [26]. For low dimensions this makes little difference, but as the number of vertices increases exponentially with the number of dimensions (2^d), this results in a dramatic increase in speed [26].

2.4.3 Operations on Q-Nets

Useful operations can be applied to Q-Nets. For the purposes of this dissertation, we shall discuss affine transformations, slicing and integration over sub-domains [26].

2.4.3.1 Affine Transformations

If the input to the neural network being integrated is transformed as $\boldsymbol{Mx}+\boldsymbol{c}$ where $\boldsymbol{M}$ is a transformation matrix and $\boldsymbol{c}$ is a translation, the neural network can be rewritten as:

$$\begin{aligned}\tilde{f}_w(\boldsymbol{x}) &= \boldsymbol{w}_2\sigma(\boldsymbol{W}_1(\boldsymbol{Mx}+\boldsymbol{c})+\boldsymbol{b}_1)+b_2 \\ &= \boldsymbol{w}_2\sigma(\tilde{\boldsymbol{W}}_1\boldsymbol{x}+\tilde{\boldsymbol{b}}_1))+b_2 \qquad (2.14)\end{aligned}$$

$$\text{Where} \quad \tilde{\boldsymbol{W}}_1 = \boldsymbol{W}_1\boldsymbol{M} \quad \text{and} \quad \tilde{\boldsymbol{b}}_1 = \boldsymbol{b}_1+\boldsymbol{W}_1\boldsymbol{c}$$

The integral of this transformed network can be found by using transformed weights and biases, divided by the absolute value of the determinant of the Jacobian of the affine transformation, in the place of the original weights and bias [26].

2.4.3.2 Slicing

If the proxy is sliced through r ($< d$) dimensions with those dimensions being 'set' to the constants $\boldsymbol{x}^{1,\ldots,r} = \boldsymbol{c}^{1,\ldots,r}$, then the neural network can be re-written as:

$$\tilde{f}_w(\boldsymbol{x}) = \boldsymbol{w}_2\sigma\left(\tilde{\boldsymbol{W}}_1 x^{r+1,\ldots,d}+\tilde{\boldsymbol{b}}_1\right) \qquad (2.15)$$

$$\text{Where} \quad \tilde{\boldsymbol{W}}_1 \equiv \boldsymbol{W}_1^{:,(r+1,\ldots,d)} \quad \text{and} \quad \tilde{\boldsymbol{b}}_1 = \boldsymbol{b}_1+\boldsymbol{W}_1^{:,(1,\ldots,r)}\boldsymbol{c}^{1,\ldots,r} \text{ [26]}$$

This removes the weights corresponding to the corresponding dimensions, and adds the values those weights would produce at the slicing constants to the bias term [26].

The integral of this dimensionally reduced network can be found as before, except the dimensionality of the integration is $d-r$ and the the dimensionally reduced weights and updated bias replace the original weights and bias [26].

2.4.3.3 Integrals Over Sub-Domains

As described in [26], to integrate over a domain [a, b], the rows of S must be updated with the new bounds of the dimensions:

$$\text{lower bound of dimension } i \text{ is } min(a^i, b^i)$$

$$\text{upper bound of dimension } i \text{ is } max(a^i, b^i)$$

and the 2^d should be replaced with $\prod(b^i - a^i)$ for all dimensions i in equations 2.10 and 2.12 (and 2.11, though this is replaced by 2.12).

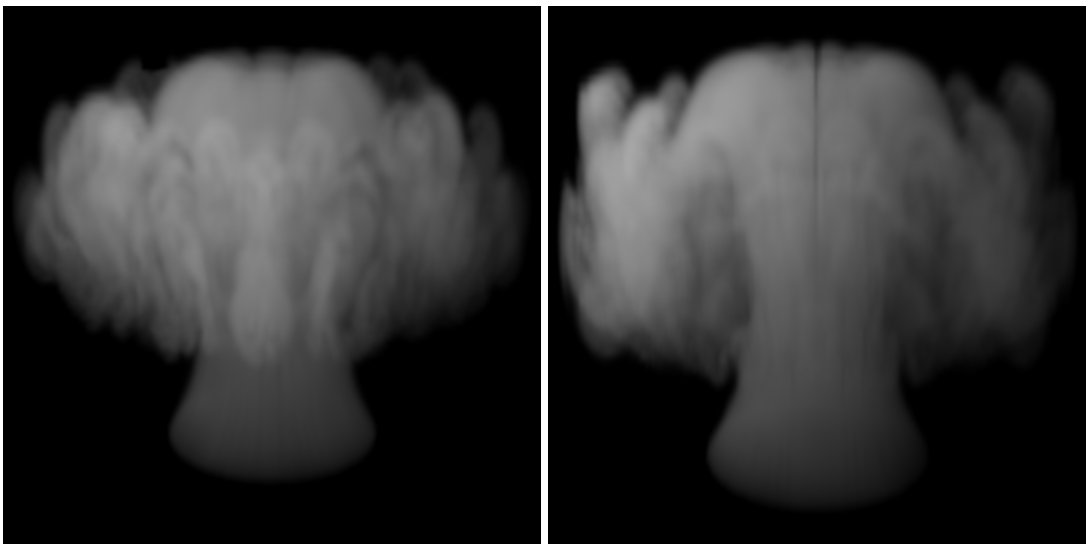
Chapter 3

Implementation

3.1 Volumetric Data Generation

We used Blender to produce a smoke simulation. Blender [6] saved the frames of the simulation in a folder within the blender project directory using the .vdb format. We chose frames from this folder arbitrarily.

Initially we tried to use the C++ implementation of OpenVDB [12] to convert the volume data in .vdb format into a .bin file, however we encountered issues with compiling with OpenVDB. Instead, we used pyopenvdb [5] to open the vdb files in Python and reduce the data to the bounding area around the active volume, resulting in a size of $117 \times 117 \times 63$ voxels. The data was then converted into a NumPy [14] array which was saved as a .npy and .mat file for later use.



(a) (b)

Figure 3.1: Blender Cycles rendering of chosen smoke simulation frame

3.2 Q-Nets

3.2.1 Initial Attempts

We had access to an example implementation of Q-Nets in MatLab [26] [19], this example demonstrated basic functions and their integrals. To aid in implementing and testing our Q-Nets we started with simply trying to recreate the results from that example.

We initially wanted to implement this project using C++ but encountered many issues when searching for neural network libraries, including issues with building the libraries and compiling with the libraries.

After these attempts, we decided again to use Python due to its extensive number of machine learning libraries, the first of which we attempted to use was TensorFlow [1].

TensorFlow provides facilities to build and train models as well as to produce custom network layers. We were able to produce networks with a single sigmoid activated hidden layer and linear output layer, but encountered issues with the in-built training methods, failing to produce networks with high enough accuracy for complicated functions (e.g. a square wave).

We manage to produce high accuracy with the simplest of the example functions as can be seen in figure 3.2, and so decided to continue to try and implement the Q-Net. TensorFlow does not provide a polylogarithm activation function, meaning we had to implement a custom activation function for the Q-Net. The Python library mpmath provides a polylogarithm implementation; however we were unable to make this function compatible a TensorFlow activation function.

The second library we attempted to use was PyTorch [21]. Doing so we ran into the same issues as with TensorFlow, however noted that Pytorch made evaluating models on the GPU simpler and so decided to continue to use PyTorch.

3.2.2 Final Implementation

3.2.2.1 Training

We were again unable to train a neural network to the required accuracy using PyTorch. Because of this, we decided to use the MatLab feedforwardnet, like from the example code from **"Q-NET: A network for low-dimensional integrals of neural proxies"**[26], to build and train the model. We then saved the model's weights into a .mat folder which we could open in python.

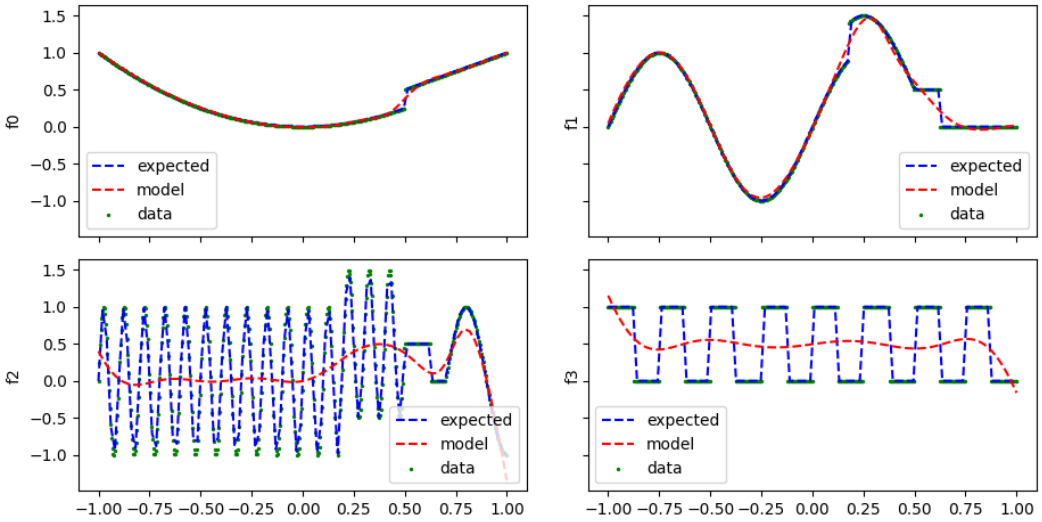


Figure 3.2: Results of example training attempts

3.2.2.2 Q-Net Implementation

As we will be using a Q-Net to integrate a 3D function along 1D rays, our Q-Net implementation did not need to be general; this allowed us to make some simplifications. As we shall always be integrating along a ray, which is one dimensional, we only need to implement Q-Nets for one dimensional functions.

We implemented two custom PyTorch modules. The first module we implemented was the first order Polylogarithm activation function, and the second was the Q-Net network as described in section 2.4.2.

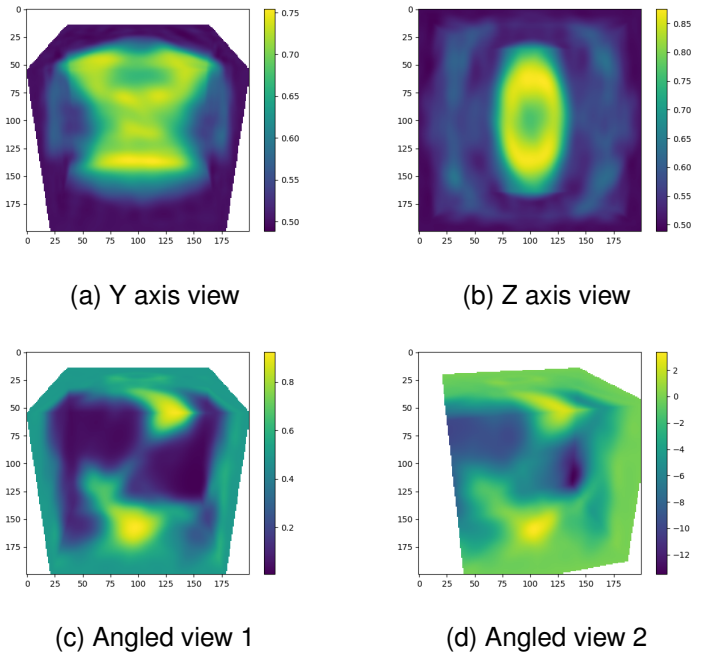
We then implemented an Integrator class. This class is constructed with the weights of the model to be integrated, storing them, and also creates and stores the Q-Net instance to be used for that model. The Integrator class implements an IntegrateRay function that takes in a ray (consisting of an origin and direction) and two parametric values for that ray indicating the points on the ray to integrate between.

Integrating along a ray occurs in 4 steps:

1. Apply a transformation to the weights of the model as described in section 2.4.3.1 translating the origin to the starting point of the ray and rotating the x-axis to align with the direction of the ray (using Rodrigues' rotation formula [3]).
2. Slice the y and z axes at the values 0 and 0 as described in section 2.4.3.2
3. Update the upper bound of the sub-domain being integrated [0, ray length] as described in section 2.4.3.3.
4. Apply the Q-Net and use its results to produce the integral as described in sections 2.4.2 and 2.4

3.2.3 Issues Encountered

We encountered issues when implementing the transformation step of ray integration. We were accidentally transforming the ray to the x-axis instead of the x-axis to the ray, this resulted in almost correct results from some views and very incorrect results from others, which made the process of fixing this error difficult 3.3.



(a) Y axis view

(b) Z axis view

(c) Angled view 1

(d) Angled view 2

Figure 3.3: Examples of renders using an incorrect transform

We also encountered problems with the MatLab feedforward net scaling the inputs and outputs to range from -1 to 1 exactly. This was not an issue for the input as we had formatted the data in that way already, however the density (expected output) ranged from 0 to approximately 0.8, as such the results from the neural network were translated and scaled. This resulted in the a need to undo this transformation after the integral was calculated, otherwise the integrals would be shifted and scaled similarly to the neural network output.

3.3 Ray Marching

In order to evaluate the Q-Net optical depth approximation, we needed to produce a ground truth image. We decided to use ray-marching through voxels in order to achieve this.

3.3.1 Voxel Traversal

(a) deltas (Blue), tmax (red)

(b) Voxel intersection illustration

Figure 3.4: Voxel ray-marching illustration

We used [2] as a basis for our voxel traversal implementation. The algorithm described in [2] operates by first finding:

- The starting voxel.
- The parametric values required to travel the width, height and depth of a voxel, called the deltas.
- The parametric values required to travel from the start point across the boundary of the current voxel for each dimension, called tmax.

The deltas and tmax values have understandable geometric interpretations, illustrated in figure 3.4a

The voxel traversal then operates by:

1. Finding the minimum tmax value and incrementing it by the corresponding dimension's delta
2. Incrementing the current voxel by 1 or -1 in the appropriate dimension according to the direction travelled.
3. Checking a stop condition, otherwise repeating

This algorithm was designed to visit every voxel along a ray, but was unconcerned with the length of the ray segment within each voxel. We adapted the voxel traversing loop to track the length of ray segment within each voxel without increasing the number of loop iterations required. This adaptation changed the first step of the loop to:

1. Find the minimum tmax value v in dimension d, decrement all tmax values v, set the tmax value used to the delta of dimension d.

The value v is the length of the ray segment within that voxel; this is illustrated in figure 3.4b.

3.3.2 Volume Marching

To use this voxel marching algorithm to calculate optical depth the volume data is read in the form of a 3D array, the size of which is used to initialise necessary parameters for marching (e.g. voxel area dimensions). Within the traversing loop, the density for each voxel weighted by the length of the ray segment within that voxel is accumulated. The total value accumulated at the end of the loop is the volume encountered by the ray.

3.4 RayTracer

The scope of the ray-tracer was limited to simply measuring the optical depth of each pixel. This measurement could later be used in order to correctly colour the pixel, as described in 2.2, however we shall not be doing so in this dissertation. Due to this limited scope, we decided to only implement the minimum of features a ray-tracer would need to do so. These features were: transforms; a pin hole camera; and bounds intersection. The implementation of these features was based off the descriptions from **"Physically based rendering: From theory to implementation"**[22].

3.4.1 Transforms

The transform class abstracts transformation matrices, the creation and the application of those transformations. The implemented transformations are:

- Translation: $T(x,y,z)$

$$M = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M^{-1} = \begin{pmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Scale: $S(x,y,z)$

$$M = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M^{-1} = \begin{pmatrix} \frac{1}{x} & 0 & 0 & 0 \\ 0 & \frac{1}{y} & 0 & 0 \\ 0 & 0 & \frac{1}{z} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- LookAt: $LA(location, up, right, direction)$

$$M = \begin{pmatrix} \text{right}_x & \text{up}_x & \text{dir}_x & \text{location}_x \\ \text{right}_y & \text{up}_y & \text{dir}_y & \text{location}_y \\ \text{right}_z & \text{up}_z & \text{dir}_z & \text{location}_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M^{-1} = \text{Calculated using M}$$

- Perspective: $P(fov, near, far)$

$$M = \begin{pmatrix} \frac{1}{tan(\frac{\pi \times \text{fov}}{360})} & 0 & 0 & 0 \\ 0 & \frac{1}{tan(\frac{\pi \times \text{fov}}{360})} & 0 & 0 \\ 0 & 0 & \frac{\text{far}}{\text{far}-\text{near}} & 1 \\ 0 & 0 & -\frac{\text{far} * \text{near}}{\text{far}-\text{near}} & 0 \end{pmatrix} \quad M^{-1} = \text{Calculated using M}$$

We also implemented the multiplication of transformations:

$$T1 \times T2 = T3 \quad \text{Where:}$$

$$M_{T3} = M_{T1} \times M_{T2} \quad M_{T3}^{-1} = M_{T2}^{-1} \times M_{T1}^{-1}$$

3.4.2 Camera

The camera class holds the camera's transforms and provides a GenerateRay function that produces a ray from the camera when given a pixel coordinate on the sensor. The transforms the camera stores are the CameraToWorld transform, simply a LookAt transformation, and the RasterToCamera transform which is:

$$\begin{aligned} \text{ScreenToRaster} = S\Big(\frac{\text{image\_width}}{\text{aspect\_width}}, \frac{\text{image\_height}}{\text{aspect\_height}}, 1\Big) \\ \times T(-\text{aspect\_left}, -\text{aspect\_top}, 0) \end{aligned}$$

$$\text{RasterToCamera} = P(\text{fov}, \text{near}, \text{far})^{-1} \times \text{RasterToScreen}^{-1}$$

Where:

- image_width is the width of the image in pixels
- image_height is the height of the image in pixels
- the aspect ratio is $\frac{\text{image\_width}}{\text{image\_height}}$
- aspect_width is $max(2, 2 \times \text{the aspect ratio})$
- aspect_height is $min(2, 2 \times \frac{1}{\text{the aspect ratio}})$
- aspect_left is $-max(1, \text{the aspect ratio})$
- aspect_top is $-min(1, \frac{1}{\text{the aspect ratio}})$

The Generate Ray function uses the RasterToCamera transform to place the desired pixel in camera space, and uses the coordinate produced to calculate the direction the ray will travel in camera space (the ray will travel from the pixel, through the pin hole). The direction is then transformed to World space using the CameraToWorld transform, normalised, and a ray starting at the camera's position with the direction just calculated is returned.

3.4.3 Bounds

The Bounds class stores the 'top right' and 'bottom left' corners of a cube and implements an intersect function that takes in a ray and returns true if the ray intersects the cube, false otherwise, as well as two parametric values for where near and far intersections were.

3.4.4 RayTracer

The ray-tracer creates single instances of the camera, bounds, Q-Net and ray marcher. To simplify the implementation, the scene consists of only a bounds instance centered at the origin with corners at (-1,-1,-1) and (1,1,1), this means that no transformation (scaling, rotation or translation) of the ray is required to format it properly for the Q-Net, placing its intersections on the unit cube.

The ray-tracer runs through all the pixels, calling the camera's GenerateRay and the bound's intersect functions for each. If the ray intersects the bound then the Q-Net (or Raymarcher) is used to calculate the optical depth of that ray, otherwise the ray-tracer moves onto the next pixel. We used MatPlotLib [16] to output the generated images, as the MatPlotLib API is simple to use.

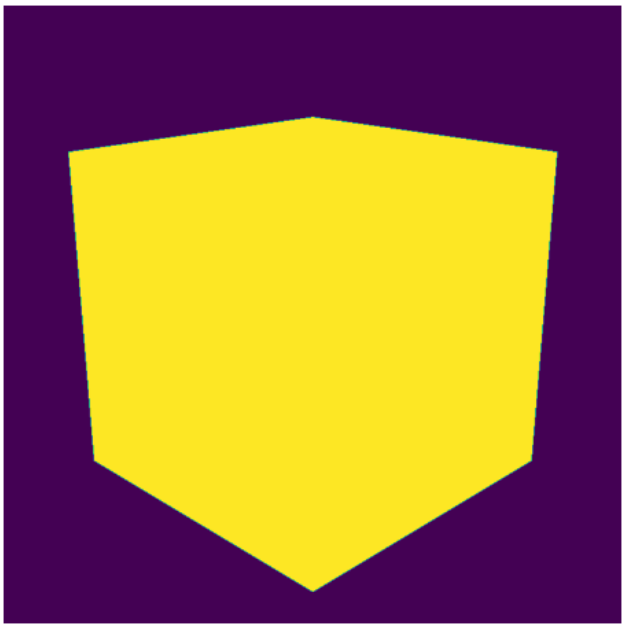


Figure 3.5: Ray-tracer view with no optical depth measurements

3.5 Threading

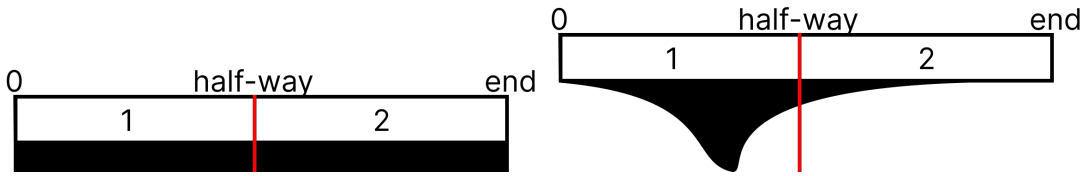
We used MPI for Python (MPI4Py) [11, 10, 9] in order to parallelize the evaluation of pixels. We implemented three versions of the threaded ray-tracer:

1. Parallelization using message passing and a static iteration distribution schedule

2. Parallelization using shared memory and a static iteration distribution schedule
3. Parallelization using shared memory and a dynamic iteration distribution schedule

For all implementations, the main worker (worker 1) initialises the integrator and ray marcher with volume data and then passed them to all other workers.

3.5.1 Static Schedule - Message Passing



(a) Static scheduling with even workload distribution

(b) Static scheduling with uneven workload distribution

Figure 3.6: Static Scheduling illustrations

Static scheduling operates by splitting the iterations (pixels) as evenly as possible between the workers. This sequential split of the work results in the workers being able to take advantage of data locality. However, if the data is unevenly distributed among the iterations, the performance gain can be minimal.

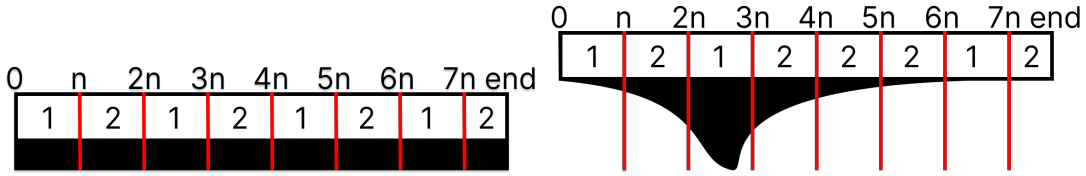
This is illustrated in figure 3.6b, where both worker 1 and 2 are allocated the same number of iterations, however the majority of the workload occurs in the first half of the iterations. As such, worker 2 will finish its iterations long before worker 1 will and will then hang (do no work) while waiting for worker 1 to complete.

For our first implementation of static scheduling, before we start rendering, we send a message to each worker containing all of the pixel coordinates for that worker to evaluate. Each worker stores its results in a local array which is then sent to the main worker to integrate into the final results.

3.5.2 Static Schedule - Shared Memory

This implementation distributes the work in the same way as its message passing implementation; however, the image being written is placed in shared memory. This made the process of writing to the image simpler to implement and eliminated the need to collate the data into an image.

3.5.3 Dynamic Schedule - Shared Memory



(a) Dynamic scheduling with even workload distribution

(b) Dynamic scheduling with uneven workload distribution

Figure 3.7: Dynamic Scheduling illustrations

In dynamic scheduling, workers take on iterations in sets of size n, starting from the first iteration. Workers that take on a set of iterations with less work will complete those iterations sooner than workers that took a set of iterations with large workloads; they can then pick up the next available iterations. This causes all the workers to be working until the job is finished, resulting in good performance when the distribution of work among iterations is uneven such as in figure 3.7b. In this way the workload is distributed more evenly instead of the iterations being distributed evenly, as happens with static scheduling.

This implementation again holds the image in shared memory to simplify the process of storing results. However, we also hold a counter in shared memory. Before rendering, each worker is allocated a block of pixels to evaluate and the counter is set to the number of workers times the block size (the first iteration that has not yet been assigned). Once a worker has finished with the pixels it has been allocated, it checks the counter. If the counter is less than the total number of pixels then the worker ‘picks up’ the next block of pixels, incrementing the counter by however many pixels it ‘picked up’. This method uses the counter to ‘point to’ the first iteration that has not yet been allocated.

Chapter 4

Results

For our experiments we produced a neural network representing the volumetric data displayed in figure 3.1. The network had 500 neurons in its hidden layer, and after training for 50 epochs (with a batch size of 1) produced a mean square error of 0.0108 (rounded to 4 decimal places). We used this network to produce optical depth approximations for the experiments.

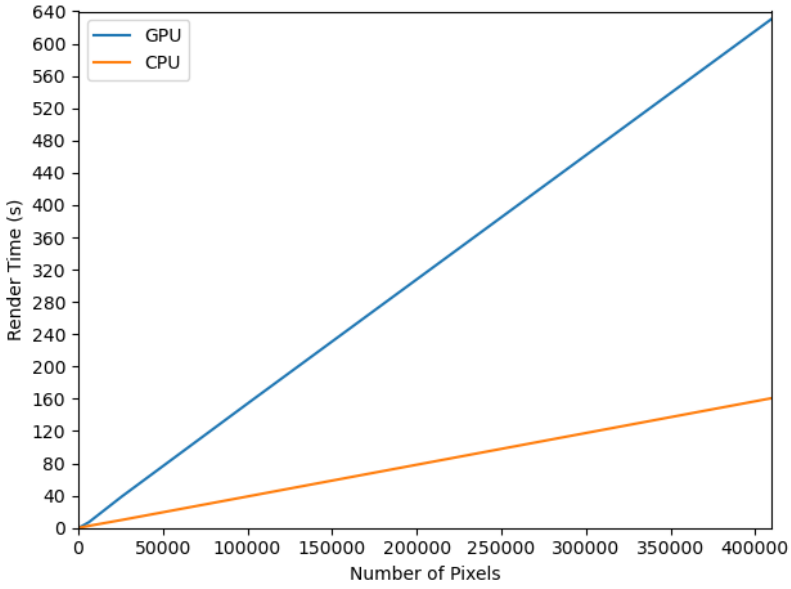
The volumetric data is $117 \times 117 \times 63$ floats voxels in size, and as such requires $117 \times 117 \times 63 = 862407$ floats in order to produce the renders. The neural network used 500 neurons in its hidden layer, as such it requires $4 \times 500 = 2000$ floats (the weights of the network) in order to calculate the integrals. This means the Q-Net implementation uses 431.2035 times less memory than the ray-marcher. This can be improved further by using few hidden nodes, though this may result in worse errors.

4.1 GPU Usage

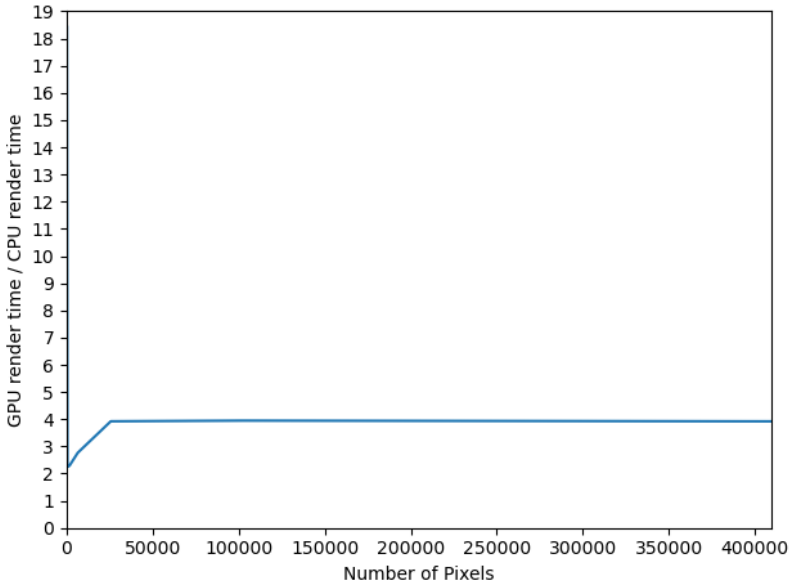
We began experimentation first by comparing the time the Q-Net rendering method took when using the GPU and when not using the GPU. We rendered the same scene 7 times, starting at a resolution of 10×10 pixels, doubling the resolution dimensions each time (increasing the number of pixels by 4).

We discovered that when using the GPU the run-time increased dramatically, as can be seen in figure 4.1. Figure 4.1a plots the run-time showing that it is related linearly to the number of pixels, the gradient of each line is the time required to calculate a single pixel. Figure 4.1a clearly shows us that pixels take longer to evaluate when using the GPU, by taking the ratio of each data point, shown in figure 4.1b we see how much longer it takes, approximately 4 times longer.

This result was not as we expected, we expected using the GPU to reduce the run-time. We believe this outcome is due the need to create transforms for each pixel; when using the GPU, we need to create the transform and then transfer it to the GPU to be used, increasing overhead. We believe that a Q-Net on a function with much higher dimensionality, running on the GPU, would experience a benefit that outweighs this overhead, but this is not the case for this application.



(a) GPU and CPU run-time vs number of pixels



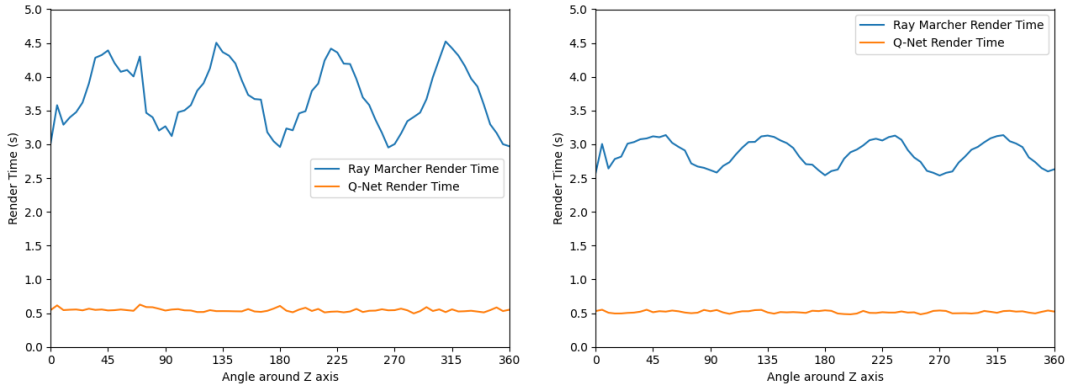
(b) Ratio of CPU to GPU run-time vs number of pixels

Figure 4.1: GPU Experiment Plots

4.2 Scheduling Method

We first wanted to find what scheduling method would be best, between the implemented static and dynamic methods, to use for our later experiments (see section 4.4). We plan to rotate a camera around each of the axis to measure how error changes with the view. As such, we reduced the number of angular samples and resolution for the experiment rotating around the Z axis to reduce the overall run-time, while maintaining the context of our later experiments. We then performed these experiments with

each scheduling method and plotted the run-time for Q-Net and ray-marching methods against the view angle.



(a) Static Scheduling method run-time vs angle around Z axis

(b) Dynamic Scheduling method run-time vs angle around Z axis

Figure 4.2: Scheduling method experiment plots

Figure 4.2 shows that the Q-Net run-time is unaffected by the scheduling method, and stays constant for all angles. However, the ray-marching run-time runs considerably faster when using dynamic scheduling.

We believe this is due to the distribution of 'work' across pixels for the the ray-marcher varies greatly (this is not the case for the Q-Net implementation) which dynamic scheduling is well designed for, as discussed in section 3.5.3. We discuss the cause of this work distribution further in section 4.4.

4.3 Error

We ran 4 experiments using this network. These experiments consisted of varying the location of the virtual camera and measuring the optical depth in the image that camera would produce using both Q-Net integration and ray-marching (using the ray-marching result as reference). We then recorded the Mean Relative Error of the pixels of each render.

$$\text{Relative Error} = \frac{|\text{measured} - \text{expected}|}{\text{expected}}$$

As the expected optical depth for some pixels was 0, we had to introduce slack into the relative error equation to prevent divisions by 0, so the equation for the relative error became:

$$\text{Relative Error} = \frac{|\text{measured} - \text{expected}|}{\text{expected} + \text{slack}}$$

While high accuracy is generally preferable, due to the nature of graphics applications, accuracy is only valuable in so far as the error is not visible to the human eye. The slack acts as an analogy for this allowance.

We ran these experiments using the dynamic scheduling implementation with 8 threads. We decided to do this as the run-time without threading would have caused the experiments to take too long.

4.3.1 Main Axes Experiments

For the first 3 experiments we rotated the camera anti-clockwise around the X, Y and Z axis starting on the positive ZY, ZX and YX planes respectively. We kept the camera at a distance of 4 and took steps of 2.5 degrees keeping the camera view centered on the origin.

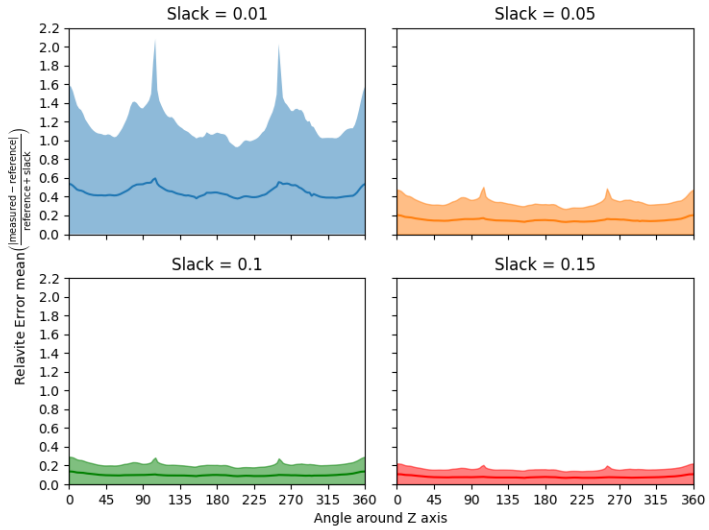


Figure 4.3: Error vs camera rotation around the X axis

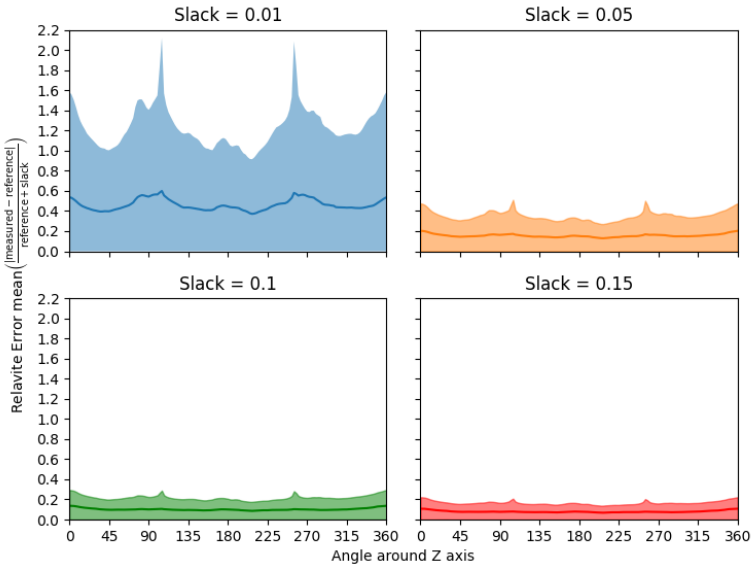


Figure 4.4: Error vs camera rotation around the Y axis

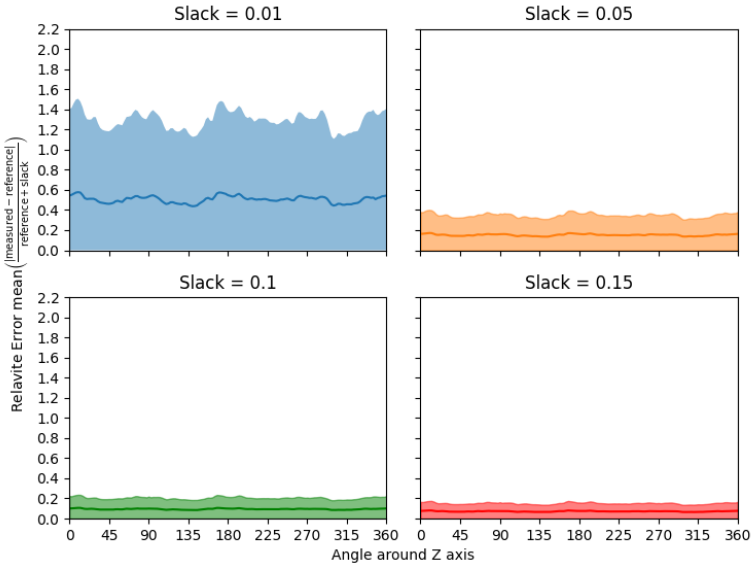


Figure 4.5: Error vs camera rotation around the Z axis

Figures 4.3, 4.4 and 4.5 plot the measured MRE (and standard deviation of that error) for each of these experiments for slack values of 0.01, 0.05, 0.1 and 0.15. As expected, as the slack value increases the Mean Error (and deviation) decreases.

The relative errors of each pixel can be used as an image to provide insight into where the errors occurred. We chose one of the flat on renders from the Z experiment (90 degree rotation) and rendered the relative errors, see figure 4.6. We chose the Z experiment as the errors had no particularly large spikes, allowing us more freedom in choosing which angle's error to render.

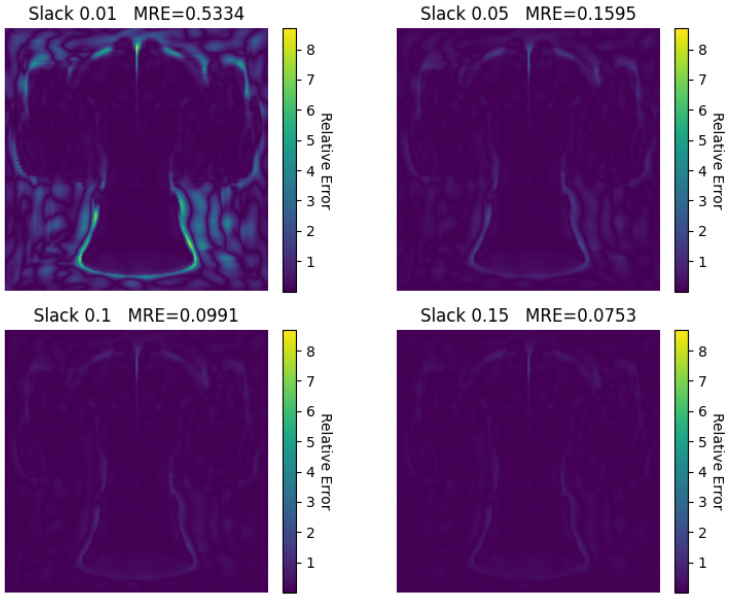


Figure 4.6: Relative error render for camera at 90 Degrees of rotation around the Z axis

The renders in figure 4.6 clearly show that the majority of error arises at the boundaries of the volume and the parts of the volume that are expected to have no optical depth at all. This is expected as the density data transitions instantly from voxels holding density to voxels with a density of 0. This will not happen in the neural network as it acts a continuous function; as such, the data will decrease more gradually producing a large error (as the expect value becomes very small very quickly).

The X and Y experiments (figures 4.3 and 4.4) experience high spikes in error variance at angles of 105 and 255 degrees from their respective planes. These views can be seen in figure 4.7. From these renders, we can see that these viewpoints cause the base of the volume to align with the rays that pass through it.

We believe this is due to the voxel nature of the ray-marcher and data combined with the high rate of change in density.

When a ray aligns closely with the axes of the voxels, it is likely to travel only through one ‘row’ or ‘column’ of voxels. As the rays corresponding to nearby pixels are similar in direction to each other, the neighbours of a pixel with a ray that aligns in this way are also likely to produce rays that align this way. This produces areas of pixels that have traversed the same ‘column’ or ‘row’ of voxels producing a uniform optical depth value across them.

This phenomena only occurs due to the use of homogeneous voxels, and as such does not appear in the Q-Net method.

This is not an issue when this occurs through areas with not much optical depth or areas where the optical depth changes very little. However, when the optical depth changes steeply between neighboring ‘rows’ or ‘columns’ of voxels then, producing a ‘blocky’ effect.

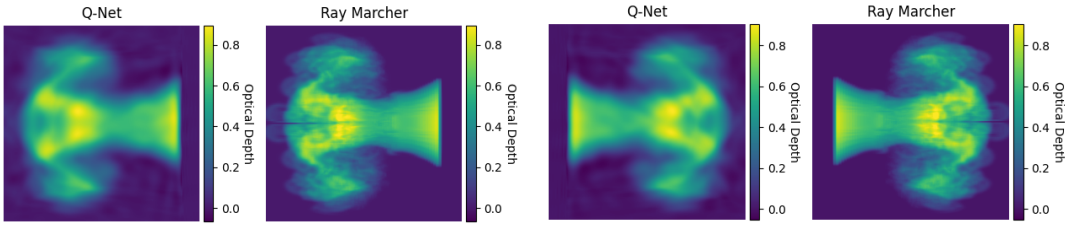
We believe this is what occurs near the base of the volume, the optical changes rapidly in the z direction, producing large differences between pixels traced through each voxel ‘row/column’.

This will also suffer from the issue around boundaries discussed below figure 4.6. These two phenomena combined are likely the cause of the spikes in error.

Implementing density interpolation between voxels would likely reduce this error; however this is out of the scope of this dissertation. This would also introduce a new problem directly bellow the volume, where the ray-marcher will interpolate density in the voxel directly underneath the base, placing more optical depth in a layer of voxels that should have little to none.

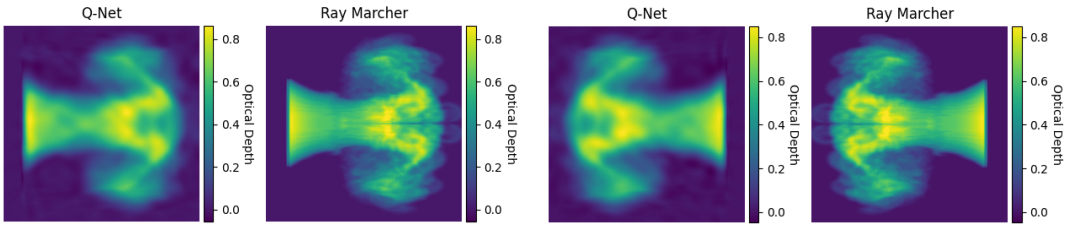
We confirmed the cause of these ‘error spikes’ by observing the lack of ‘error spikes’ in the Z axis experiment (figure 4.5), whose view never aligns with the base of the volume, as well as in the fourth experiment. Additionally, figure 4.11a shows the relative error of each pixel from a similar spike in error in the 4^{th} experiment. From this figure, it is clear to see that the spike in error occurs directly below the base of the volume, as we predicted. The the relative error here reaches 30 (3000%), while the error in the rest of the render stays within the same range as other renders though figure 4.11b shows that the absolute error is never greater than 0.4.

We also ran the Y axis experiment for a volume that does not have the harsh drops in density we predict causes the 'error spikes'. This volume can be seen in figure 4.8a and the results of the experiment can be seen in figure 4.8b. As we predicted, the error plot in figure 4.8b does not have any 'error spikes' as the volume lacks the harsh density drops described.



X Experiment Y Experiment

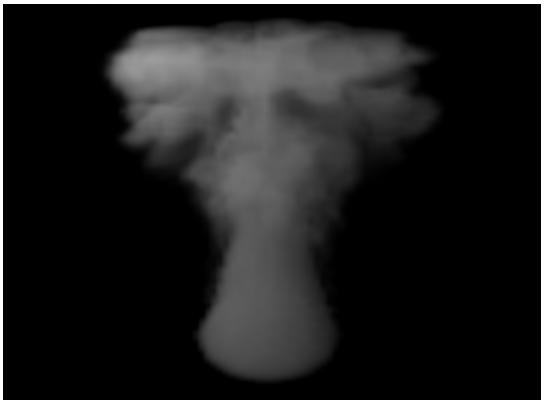
(a) 105 Degrees from start



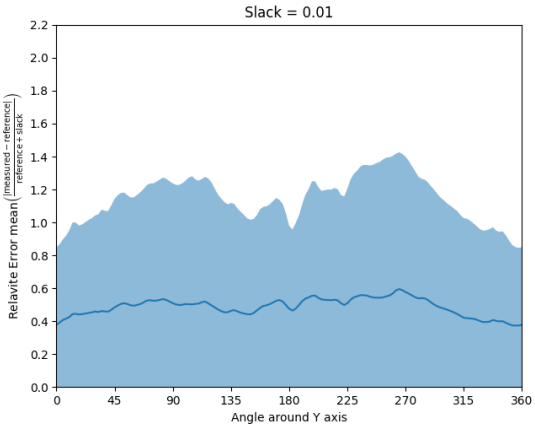
X Experiment Y Experiment

(b) 255 Degrees from start

Figure 4.7: Error Spike Renders



(a) Secondary Volume Render



(b) Error Plot around Y axis

Figure 4.8: Secondary Volume Experiment

We do not believe this special case is a problem in practice due to the nature of volumes like this. Clouds effectively never have situations where density transitions from high to nothing in a single voxel boundary. Smoke simulations like the one used as an

example do however. In any situation where a smoke simulation such as this is being rendered, it is very likely that an 'emitter' will be obscuring the base of the simulation.

As the Q-Net method is integrating a neural network that represents the data, training the neural network better will result in lower optical depth errors. Perfecting the neural network training was not the aim of this dissertation and doing so would have caused the work to extend beyond the deadline of this dissertation.

4.3.2 Combined Rotation Experiment

In the last experiment we ran, we rotated the camera around the Z axis at a distance of 4 units, we also varied the angle from the XY plane in a sinusoidal fashion between +45 and -45 degrees, this angle is plotted in figure 4.9.

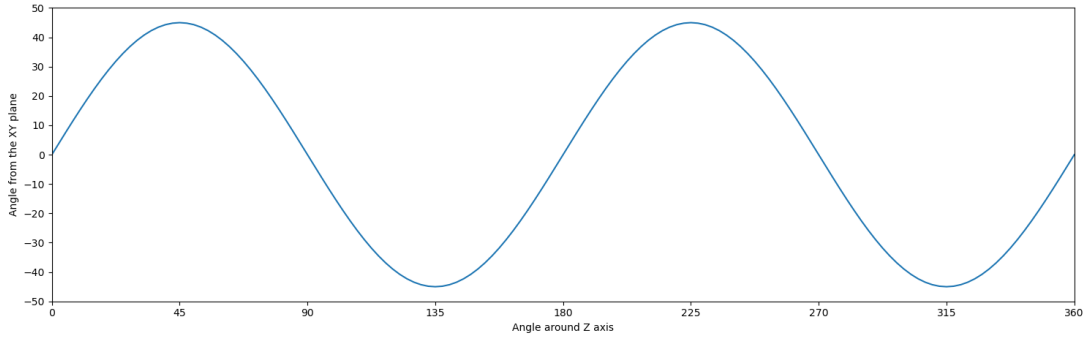


Figure 4.9: XY Angle vs Rotation around the Z axis

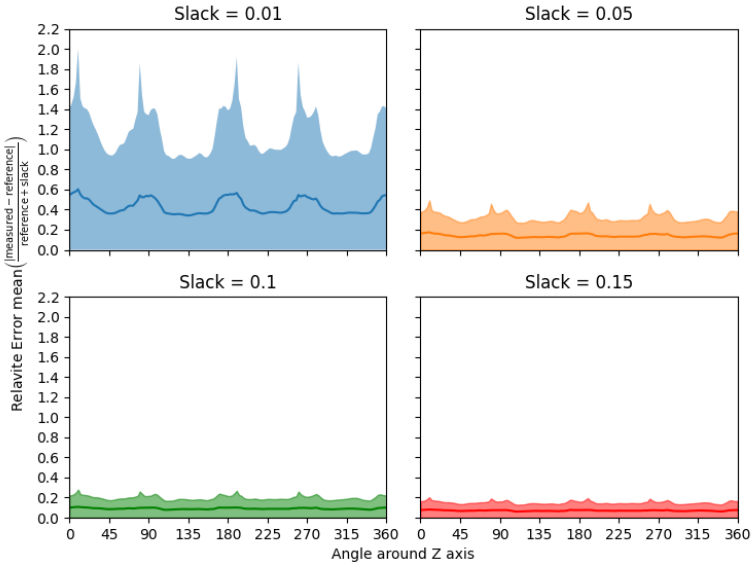
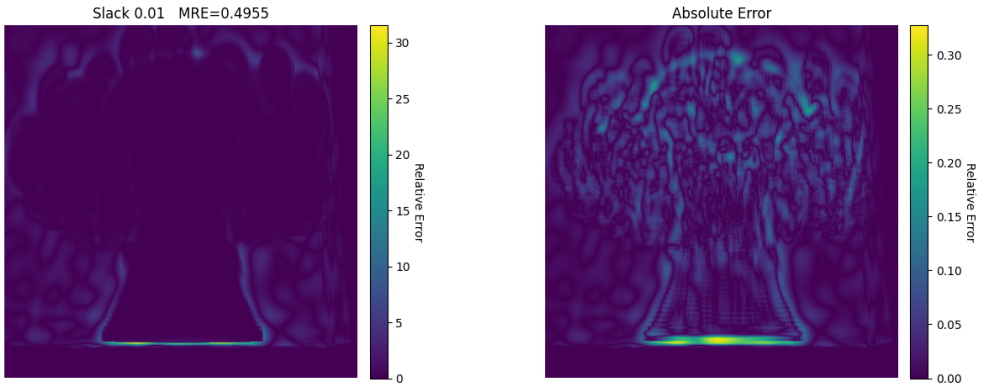


Figure 4.10: Error vs rotation around the Z axis + height modulation

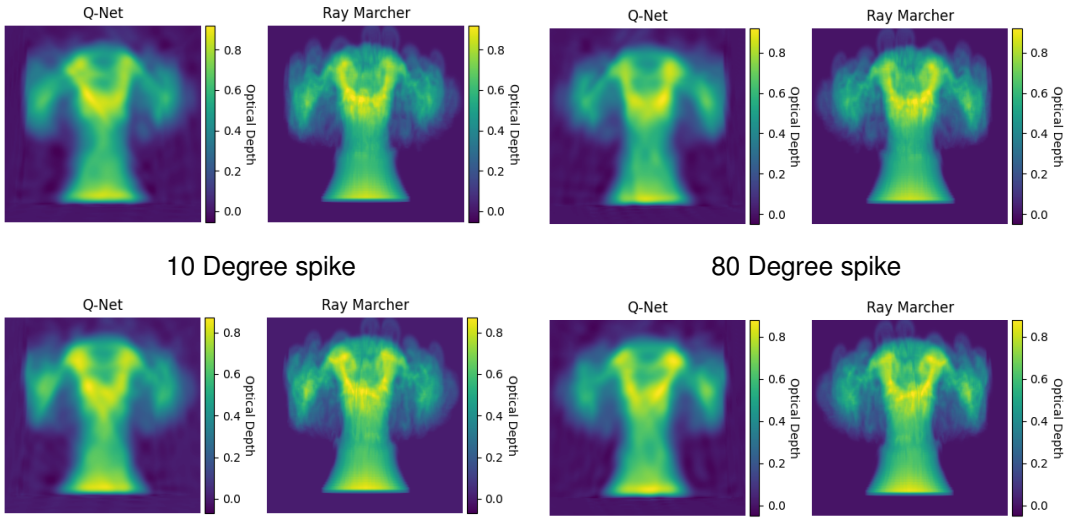
We can again see 'error spikes' in figure 4.10; the renders associated with these spikes can be seen in figure 4.12. Once again, these spikes occur with instances where the

base of the volume aligns with the rays travelling through it, giving more weight again to our hypothesis in section 4.3.1.



(a) Relative Error (b) Absolute Error

Figure 4.11: Error Renders of 80 Degree spike



10 Degree spike 80 Degree spike

190 Degree spike 260 Degree spike

Figure 4.12: Z axis + height modulation Error Spike Renders

4.4 Timing

We also timed each render used in the experiments described in section 4.3. Figures 4.13, 4.14, 4.15 and 4.16 plot the time each rendering method took to compute its respective result for each frame.

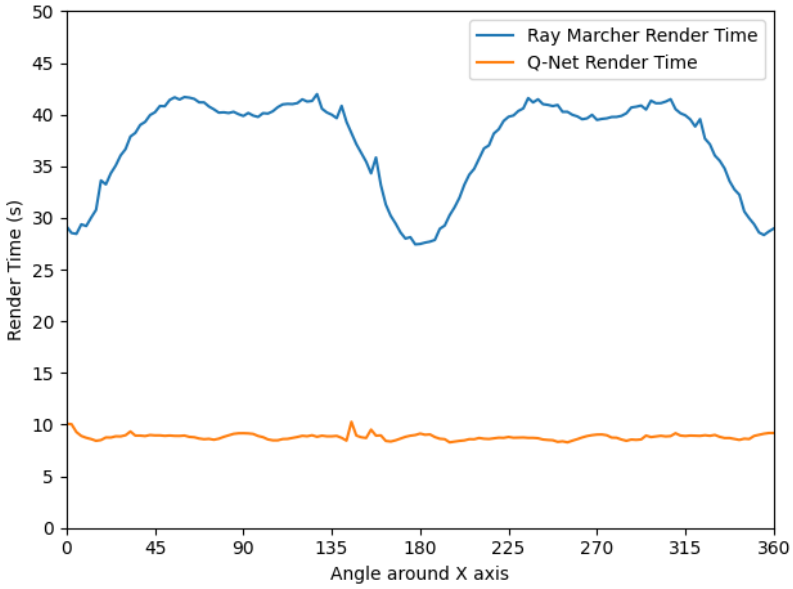


Figure 4.13: Render time vs rotation around the X axis

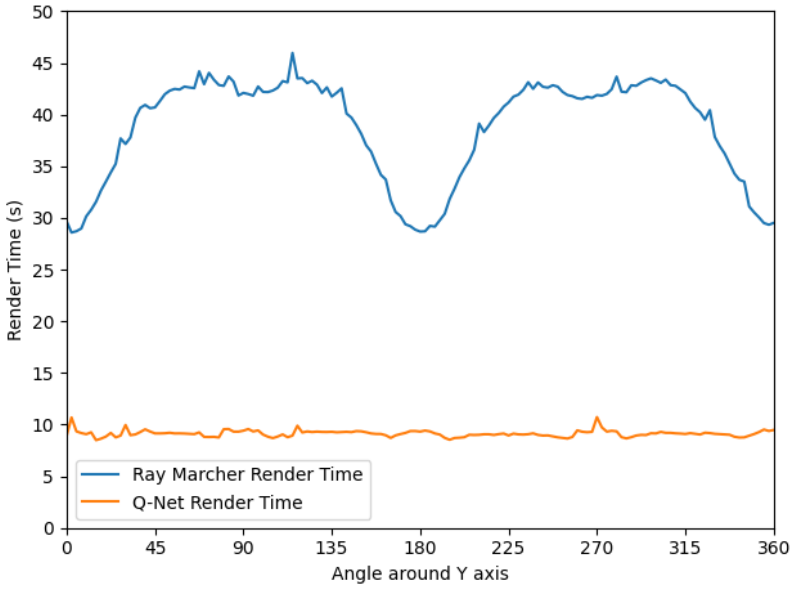


Figure 4.14: Render time vs rotation around the Y axis

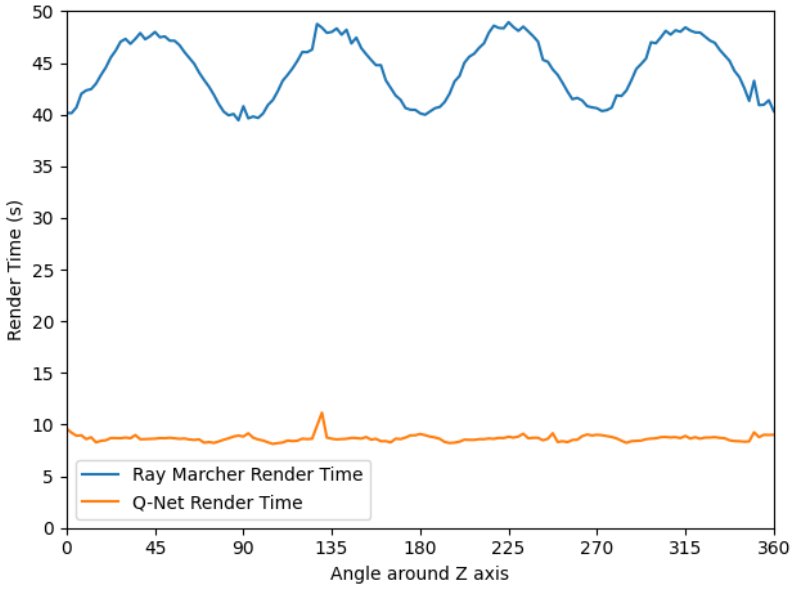


Figure 4.15: Render time vs rotation around the Z axis

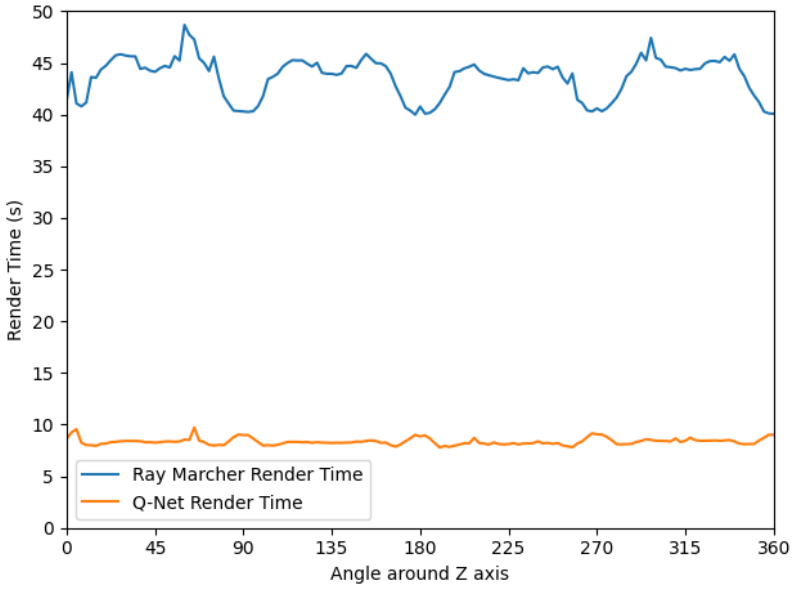
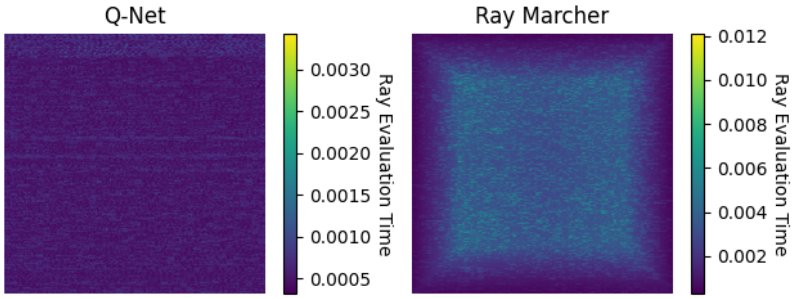


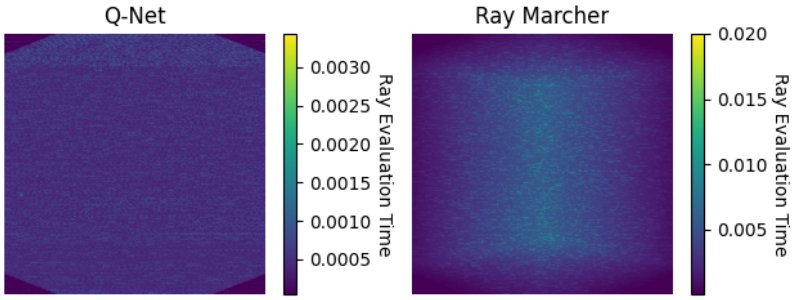
Figure 4.16: Render time vs rotation around the Z axis

We can see from these plots that the run-time for the Q-Net method's run-time stays reasonably constant at, or below, 10 seconds per frame. However, the run-time of the ray-marching method's varies considerable (by 10-15 seconds) as the view of the camera changes. These plots clearly show that the Q-Net renderer runs considerably faster than the traditional ray-marching method.

We noted the worst and best run-times of the Z experiment and re-ran those specific renders using the time each pixel's ray took to calculate as that pixel's value.



(a) 90 Degree run-time distribution



(b) 45 Degree run-time distribution

Figure 4.17: Run-time distribution from Z experiment

4.4.1 Ray-Marcher Timing Analysis

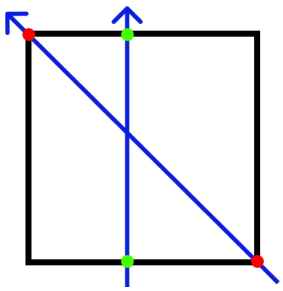


Figure 4.18: Visualisation of Ray Bounds Intersections

A ray that intersects one of the bounding cube's faces at right angles will exit the bounds at the opposite face having travelled exactly the length of the bounds in that dimension. A ray that intersects one of the bounding cube's corners and exits through the opposite corner will have travelled considerably further through the volume than the ray described before, this concept is visualised in figure 4.18. The distance through the volume a ray travels impacts the number of voxels the ray will intersect and therefore how long that it takes to march that ray.

We believe this is the source of the variability in the ray-marching time. As the camera rotates around the volume, the distance each of the rays it casts travels through the volume changes. It is clear that the view that minimises these distances is the view

flat on to the volume, which is reflected in the figures 4.13, 4.14 and 4.15 having the minimums at multiples of 90.

This is also supported by figure 4.17, figure 4.17a shows the run-time distribution for a view flat onto the volume bounds, figure 4.17b shows the run-time distribution for a view looking at an edge of the bounds. The colour bars from these figures show that the rays marched through the edge of the bounds took almost 2 times as long as the rays marched from one face to its opposite.

In the experiments as we are only rotating around one axis at a time, the view that will maximise this distance is when the camera is looking through the edges of the bounds, which is reflected in the plots having maximums at multiples of 45.

It is important to note the X and Y axis experiments do not return to the global minimums at 90 and 270 degrees, this is due to the dimensions of the voxels. The bounds are split into $117 \times 117 \times 63$ voxels, this makes rendering from views of the X and Y axes more expensive than rendering from the Z axis views.

Comparing figures 4.13 and 4.14 to 4.15 reveals that the local minimums at 90 and 270 degrees in 4.13 and 4.14, match the global minimums in 4.15. This is what we expect to see as 90 and 270 degrees from figures 4.13 and 4.14 correspond to the same views as the minimums from figure 4.15

This can be improved by pre-processing the volume data to produce tighter bounds, minimising number of empty voxels that need to be traversed, potentially eliminating the need to render some of the pixels that encounter no density. However ‘longer’ rays will result in longer rendering times.

4.4.2 Q-Net Timing Analysis

Figures 4.13, 4.14 and 4.15 show that the run-time of the Q-Net renderer does not vary like the ray-marcher. This is as we expected, as explained in section 4.4.1, the variance in ray-marching run-time arises due the need to traverse the rays and the varying length of those rays; however this is not a problem for the Q-Net method. The Q-Net method’s execution is invariant in terms of integral domain, no more calculations are required for a ray of length 2 units, and ray of length 1 unit.

This again supported by figure 4.17 where the distribution of run-time within the Q-Net rendered volume is indistinguishable from noise. This implies that the specifics of the ray have no impact on the time it takes to measure to optical depth it encounters.

We can reason that the run-time of the Q-Net rendering method, depends only on:

1. The number of neurons in the network used to represent the volume.
2. How many pixels require the Q-net to be evaluated.

This makes the Q-net method’s run-time independent of the complexity of the volume data (assuming constant neuron number) and view independent within sets of views that cause the bounds to intersect with the same number of ray casts.

Issue 1 could be improved by optimising the network training, attempting to minimise the number of neurons used. Issue 2 can be improved using the same method used to improve the ray-marching, this method will result in the Q-Net being called for fewer pixels. However care should be taken to balance the complexity of the bounds with the run time as the time to check ray-bounds intersection increases as the complexity of the bounds increases. Care will also need to be taken to ensure the intersection points (and ray directions) are transformed into the volume space correctly (should be trivial using object space transforms).

Chapter 5

Conclusions

5.1 Findings

In this dissertation we have shown that Q-Nets can be used to approximate optical depth. We have also shown that using Q-Nets to approximate optical depth has key benefits over the traditional ray-marching method:

- The Q-Net implementation is faster to evaluate
- The Q-Net implementation run-time is invariant to the distance over-which the optical depth is to be approximated.
- The Q-Net implementation run-time is invariant to the complexity of the volume data. Depending only on the number of neurons used in the neural network that represents the data.
- The Q-Net implementation is invariant to the direction of the ray, this is an extension of 2.

We also showed that the Q-Net method can be distributed across threads in order to reduce the overall run-time. We experimented with both dynamic and static scheduling and found little difference in performance gain between them. However we believe this was due to the volume consuming the majority of the camera view, if this weren't the case, we believe dynamic scheduling would have provided better performance than static, due to the uneven distribution of Q-net calls that would be required, this difference is not unique to the Q-Net implementation.

We also showed that utilising the GPU to evaluate the Q-Net yielded worse run-time than otherwise, though we believe this was due to constant need to transfer data to and from the GPU.

We discussed how the Q-Net's accuracy depends on the accuracy of the neural network being used: If the neural network has reproduced the data inaccurately, the optical depth measured will also be inaccurate. Though we discussed allowing 'slack' in the error due to the graphical nature of this application.

Finally we briefly discussed the memory usage improvements of using the Q-Net implementation: the Q-Net only needs to hold the neural network's weights in memory, whereas the ray-marcher method needs to hold the whole volume data-set.

5.2 Future Work

We would be interested in future works to:

- Develop or utilising libraries to make evaluating the Q-Nets on the GPU more performant.
- Render multiple frames: A neural network could be trained on multiple frames of a smoke simulation with time as an extra dimension.

  Once the network has been transformed (carefully leaving the time dimension unchanged), the y and z dimensions would be sliced as before. Except, the integral could be evaluated many times with the time dimension sliced at each frame.

  With only 1 or two frames of animation, the performance gain over rendering frames separately with Q-Net would be minimal. However, when rendering many frames, this would eliminate repeated transformations, which we believe to be the most time consuming element of the Q-Net method: Transform once, evaluate optical depth for the ray for multiple frames.

  This could also benefit more from using the GPU,

  1. The weights of the Q-Net would be the same for each frame sample, potentially allowing the different frame values to be evaluated in parallel.
  2. The transformed network could be sent to the GPU once, along with the time values to be evaluated, and then all the results could be returned at once (reducing the total number of CPU-GPU data transfers).
- Develop techniques to more accurately fit the required form of neural network to volume data. This could take advantage of common traits of volumes/clouds such as how volumes tend to have the highest volume in their 'cores' and have density peter off at the edges.
- Use techniques to produce more accurate bounds for the volume in order to speed up the volume rendering. Finding a 'sweet-spot' between bounds complexity and performance gain.

Additionally, figure 4.11 demonstrated the possible uses of Q-Net volume rendering for identifying specific areas of error in neural networks. We'd be extremely interested to see if this could be used as a tool to understand the workings of neural networks as well.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] John Amanatides, Andrew Woo, et al. A fast voxel traversal algorithm for ray tracing. In *Eurographics*, volume 87, pages 3–10, 1987.

[3] Serge Belongie. Rodrigues' rotation formula.

[4] Antoine Bouthors. *Rendu réaliste de nuages en temps réel*. Theses, Université Joseph-Fourier - Grenoble I, June 2008.

[5] Alex Braun. pyopenvdb, Mar 2020.

[6] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[7] Robert L Cook, Thomas Porter, and Loren Carpenter. Distributed ray tracing. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 137–145, 1984.

[8] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[9] Lisandro Dalcin, Rodrigo Paz, and Mario Storti. Mpi for python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[10] Lisandro Dalcin, Rodrigo Paz, Mario Storti, and Jorge D'Elia. Mpi for python: Performance improvements and mpi-2 extensions. *Journal of Parallel and Distributed Computing*, 68(5):655–662, 2008.

[11] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124–1139, 2011.

[12] Academy Software Foundation. openvdb, Oct 2021.

[13] Giovanni Giusfredi. *Geometrical Optics*, pages 159–309. Springer International Publishing, Cham, 2019.

[14] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[15] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[16] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[17] James T. Kajiya. The rendering equation. *Seminal graphics*, page 157–164, 1998.

[18] Steffan Lloyd, Rishad A. Irani, and Mojtaba Ahmadi. Using neural networks for fast numerical integration and optimization. *IEEE Access*, 8:84519–84531, 2020.

[19] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010.

[20] John McGonagle, George Shaikouski, Christopher Williams, Andrew Hsu, Jimin Khim, and Aaron Miller. Backpropagation.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[22] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[23] Raúl Rojas. *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[24] Alfred Schmitt, Heinrich Müller, and Wolfgang Leister. Ray tracing algorithms — theory and practice. In Rae A. Earnshaw, editor, *Theoretical Foundations of Computer Graphics and CAD*, pages 997–1030, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[25] Andrew Schneider. The real time volumetric cloudscapes of horizon zero dawn. *Siggraph*, May 2015.

[26] Kartic Subr. Q-net: A network for low-dimensional integrals of neural proxies. *Computer Graphics Forum*, 40(4):61–71, Mar 2021.

[27] Magnus Wrenninge. *Production volume rendering: design and implementation*. AK Peters/CRC Press, 2019.

[28] Zeng Zhe-Zhao, Wang Yao-Nan, and Wen Hui. Numerical integration based on a neural network algorithm. *Computing in Science Engineering*, 8(4):42–48, 2006.