Global Illumination using Photon Mapping

Stuart Jones

Bachelor of Science in Computer Science with Honours
The University of Bath
May 2013

Global Illumination using Photon Mapping

Submitted by: Stuart Jones

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

Global illumination aims to simulate the light transfer of a scene .This project documents the development of a system that performs the photon mapping algorithm with the aim of approximating the global illumination of a scene. We include an overview of the history of global illumination and the techniques that have been developed to simulate it. We then develop a design for a system that will perform photon mapping and detail the implementation including any challenges that were faces during the implementation. We conclude with a description of testing performed and provide example images from the system to showcase its capabilities.

Contents

List of Figures

Acknowledgements

Thank you to Brian Wyvill for his support and guidance throughout this project and to my family for their continued support throughout my degree.

Chapter 1

Introduction

Computer graphics is a sub field of Computer Science that has obvious appeal, that of creating interesting images with computers. Throughout this century and the latter half of the last researchers have strived to develop algorithms that simulate how the real world works and use these algorithms to produce images that are visually appealing and that appear to be in some way realistic.

1.1 Global Illumination

As the use of computers has become more prevalent in everyday life so has its use in the arts, a leading area of research is computer generated imagery we the aim of producing images that simulate the way in which light interacts with the world, global illumination techniques aim to achieve this through evaluation of the light transport equation of a scene. A true solution to this equation is computationally unfeasible as a result techniques have been developed that estimate the contribution due to indirect lighting that seek to reduce the computational cost that give a correct solution to the rendering equation at the limit, one such technique is photon mapping, this is the focus of this project.

1.2 Aims

This project aims to create a system that will synthesis images using the photon mapping algorithm to estimate the global illumination of a scene, in particular we aim to simulate lighting phenomena such as colour exchange between diffuse surfaces, caustics caused by secular objects. We aim to create a system that is by some measure efficient and that is easy to use.

1.3 Structure

This document will begin with a description of the research surrounding my project that we have used to draw from. We will then move onto the requirements for the system then a description of the design and then the implementation of the system including aspects of the implementation that were interesting or presented a particular challenge. Finally we shall provide a brief description of the testing performed on the system.

Chapter 2

Literature Survey

In this chapter we shall discuss background literature relating to the aims of this project as having a good understanding of the work of others and the motivations for their work will provide a basis for the work undertaken in the project.

2.1 Flux

The photon mapping algorithm is based around emitting photons from light sources, the unit of measurement of a photon is Radiant flux Φ is a measurement of radiant energy with respect to time (**?**), the SI unit of flux is the watt (W). Flux is a spectral phenomena with its energy being distributed in different wavelengths, in this project we will only consider three wavelengths red, green, and blue.

2.2 Radiance

Radiance commonly refers to two quantities, incident radiance and exitance radiance, incident radiance being the radiance that fall on a surface, exitance radiance being radiance from a surface (either from reflection or emission). Radiance is a measure of the radiant flux that falls within a given solid angle in a direction above a surface. Radiance is commonly given the symbol L_i for incident randiance and L_o for exitance radiance.

2.3 The BDRF

Global illumination systems need to describe the radiance from an object, this includes the reflected radiance by an object from the scene. The bidirectional reluctance distribution function (**?**)is a function that describes the reluctance of a surface at point x with respect to an outwards direction ω_o, and inwards direction ω_i. The BDRF is commonly written as

$f_r(x, \omega_i, \omega_o)$. The BDRF is a vital part of computer graphics as it describes the reluctance of a surface which is vital in global illumination as we need to consider reluctance from all objects in a scene.

The BRDF is a me sure of the outgoing radiance L in a given direction ω_o from an incoming irradiance E from a direction ω_i. The BRDF for a surface is given by:

$$f_r(\omega_i, \omega_o) = \frac{L_{\omega_i}}{E_{\omega o}} \tag{2.1}$$

An example of a simple BDRF would be that of Lambertian reflection whereby the light is reflected in all directions equally, the BRDF for this is:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi} \tag{2.2}$$

Where ρ is the reflectively of the surface, for physically based rendering this is a real number between zero and one, the denominator in the expression ensures that the amount of energy reflected by the surface equals the amount incident on the surface.

2.4 The Rendering Equation

First described by Kajiya the rendering equation is an formulation to the light transfer of a scene (**?**). A frequency independent version of the rendering equation is given below:

$$L_r(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i \tag{2.3}$$

where L_r is the radiance from the surface at point x in the direction ω_o , L_e the emitted radiance from the surface and f_r is the BDRF of the surface at x as described in section 2.3.

There have been many developments within computer graphics that attempt to find approximations to the rendering equation, radiosity (**?**)which attempts to find the solution by splitting the geometry of a scene into smaller patches and builds a system of linear equations that solve the radiosity value for the patch, this technique is simple but sufferers from some problems, for one it is only able to account for Lambertian diffuse reflections from other object and so cannot be used to render specular reflections. Radiosity is still used within architectural applications as the radiometry is view independent and as such is well suited to applications such as walk-throughs. Monte-carlo methods are another method that is more general that radiometry, this method is a stochastic method that attempts to find the solution of the rendering equation by sampling the light transfer of a point until a adequate approximation is found, a common form of this is distributed ray-tracing whereby ray-tracing as introduced by Whitted (**?**) is performed but for each intersection the choice of reflecting, refracting or absorbing is taken from a distribution as the number of rays emitted increases the image converges to the solution of the rendering equation.

Understanding the motivations behind the development of the rendering equation will be vital if we are to be successfully in developing a photon mapping system, the solution that most rendering algorithms produce are an approximation to the rendering equation and as such will be the focus of my system.

2.4.1 Monte Carlo Methods

The rendering equation describes the light transfer as an integral of all directions above the point being evaluated, this integral cannot be solved with a closed form solution, as a result we must approximate the value of the integral, to do this we use monte-carlo methods (**?**). Monte carlo methods allow us to estimate the value of an integral at a point. Given a function $f(x)$ which we would like to integrate over the range $[a, b]$

$$I = \int_a^b f(x)dx \tag{2.4}$$

we can form a monte-carlo estimate I_m by taking the average of N uniform samples $\xi_0 ... \xi_n$ over the range $[a, b]$

$$I_m = (b - a)\frac{1}{N}\sum_{i=1}^{N} f(\xi_i) \tag{2.5}$$

it can be shown that as N tends to infinity our estimator converges to I

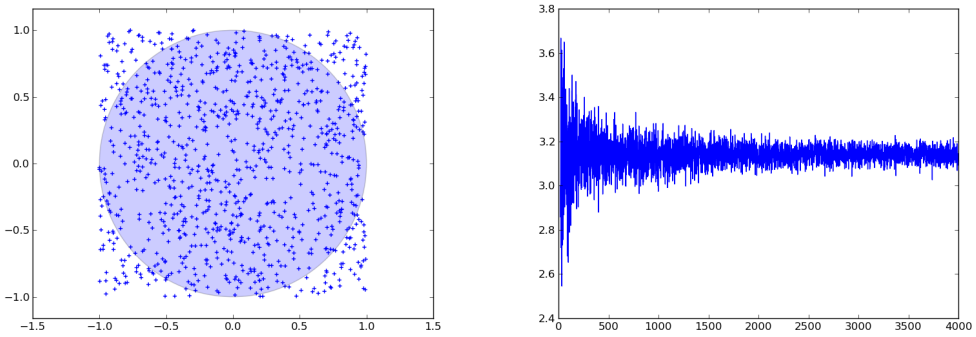
$$\lim_{N\to\infty} I_m = I$$

An Example

To illustrate the use of monte-carlo method we shall use it to approximate the well known value π, to do this we take n samples (ξ_0, ξ_1) on a square with area 4, we then calculate the ratio c / m where c is the number of samples where $\xi_0^2 + \xi_1^2 <= 1$, this will approximate the area of the square and a circle with radius 1 which has closed form solution $\pi/4$, as we increase the number of samples we approach this value, this can be seen in Figure 2.1.

2.5 Ray-tracing

Ray-tracing is a technique with early developments by Appel (**?**) and improvements by Whitted (**?**) that aims to produce images from scene descriptions that mimics the way in which light acts, ray are emitted from a virtual eye into a scene and the closest intersection with the ray in the scene is found, the algorithm then calculates the colour of the object



(a) Samples for the simulation

(b) Convergence as n increases

Figure 2.1: Monte Carlo simulation to approximate π

at the point of intersection, this may include refractive or reflective materials that require additional rays to be traced through the scene, this is called recursive ray-tracing, certain effects can be simulated well with this technique such as shadows which are formed by checking the visibility of the point of intersection with the light sources in the scene, if the light is blocked by an object in the scene the object does not receive light from that light source.

2.6 Path Notation

The radiance estimate on the right hand side of the rendering equation is typically evaluated by a recursive call to the procedure that evaluates radiance at a point in the direction ω', as we perform more evaluation we create a path from the eye to the final destination of the ray, it is frequently convenient to refer to these paths by the interaction that the ray has with surfaces along the path, we use a regular expression language to express these paths, tokens in the language include:

E The eye or camera

D A diffuse surface

S A specular surface.

L A light

Paths begin with the eye and end at a light, a full solution to the rendering equation will evaluate the radiance due to all paths **E(S|D)*L**, ray-tracing evaluates all paths **ES*DL**

2.7 Photon mapping

Photon mapping was first developed by Jensen (**?**) as an extension to the traditional ray tracing algorithm proposed by Whitted (**?**) whereby a prepossessing step is added that creates a photon map of the scene.

The photon mapping algorithm begins by emitting photons from each light source in the scene, each photon contains a fraction of the power of the light that is proportional to the number of photons that are emitted from the light source. Once emitted from the light source the photon is traced through the scene much like a ray in ray-tracing, the photon is absorbed by a diffuse surface within the scene, each of the photons absorbed are stored within a photon map which is an efficient data structure that will store information about the photon such as the location of the absorption, the direction from which the photon was travelling and the photons energy. the photon map is a spatial structure as we will query the photon map for the photons within a location as such it is common to store the photon map in a kd-tree to allow an efficient nearest neighbour search. Each photon emitted from the light source can be stored multiple times along its path in this way we can use the photon map to estimate the radiance from all light paths connecting the light and the eye.

Multiple photon maps are used with different accuracies in order to improve the efficiency of the algorithm, for instance in (**?**) two photon maps are used, a global photon map and a caustic photon map, the caustic photon map contains photons that arrive at a location from either the specular reflections or refractions , this photon map is stored at a higher accuracy as its data will be visualised directly to produce the appearance of caustics in an image. Storing the caustic causing paths in a separate photons not only makes the simulation of caustics more efficient it also reduces the variance in renderings as the density from all other paths tend to vary more slowly (**?**).

The general photon map contains all other types of photons stores, such as direct and indirect illuminations and shadow photons as described in (**?**; **?**) these shadow photons allow for the number of shadow rays to be reduced as we do not need to shoot shadow rays if the photon search returns only shadow photons.

The second stage in the photon mapping algorithm is the ray-tracing stage, this state is similar to the algorithm described in the section above but we now have the data stored within the photon map to make use of. Rays are shot from the eye into the scene until an intersection is found, we then use the properties of object hit and the photon maps to calculate the colour set for the pixel, this is done by estimating the radiance at the point of intersection, a nearest neighbour search is performed on the photon map centred at the point of intersection, the radius of the search is increased until the number of photons has reached a predetermined number n, the estimate of the radiance is then given by the sum of the radiance's of the photons within the search.

$$L_r(x, \vec{\omega}) \approx \sum_{p=1}^{n} f_r(x, \vec{\omega}'_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}'_p)}{\pi r^2} \tag{2.6}$$

The radiance is estimates as the density of a disk of radius r, this estimate is used as it is assumed that the photons within the search will lie on the surface of the object and locally relatively flat, this is accounted for by the division of πr^2 term in equation 2.6.

The density estimate given in (**?**) can produce poor results in the case of geometry with areas that are close together but not flat, such as the corners of a room, in this case the estimate of the radiance will be too high as the photons lying on the other wall will contribute to the estimate, a solution to this is given in (**?**), where the search radius is flattened along an axis orthogonal to the normal of the surface.

2.8 Participating media.

Participating media are materials or objects that interact with light within the surface of the object, these interactions can create interesting visual effects that add to the realism of a scene, a good example of participating media would be smoke, as light enters smoke it will experience scattering that will cause the path of the light to be altered. (**?**) In order to render participating media a solution to the volume rendering equation must be found, this equation describes the light transfer within the participating media, Jensen presents a good explanation (**?**).

Participating media are generally split into two types, inhomogeneous and homogeneous, this describes if the properties of the participating media are constant, an example of a participating media that is homogeneous would be a liquid such as milk, an example of a non-homogeneous media would be smoke as the density of the smoke (and as a result how much it scatters light) is not constant.

Early work in this area include work done by Blinn (**?**), in this paper a model for the scattering in clouds and other such media is given, this paper only considers single scattering in its approximation and as noted in (**?**) only suitable for optically thin media.

2.8.1 Volume ray casting

Volume ray casting is a technique that is central to participating media as it allows rays to be seen as moving through a media, this is apposed to traditional ray-tracing where the ray will only interact at the point of intersection of an object. In ray casting the ray is "marched" through the medium, at each step in the process a calculation can be performed in order to decide if an action should be taken (i.e in or out scattering)

2.8.2 Volumetric Photon Mapping

One of the earliest applications of photon mapping to render participating media was (**?**) in this paper the concept of volumetric photon mapping and the volume photon map was introduced, this photon map allows the interaction of light within a participating media

to be taken into account, this could be one of the following, in-scattering, out-scattering, absorbs ion or emission. The volume map is a separate map to the surface map usually used in the photon mapping algorithm, this is because the density estimate is different as the sample of photons are taken from the volume enclosing n photons and not the disk. The technique in this paper deals with the case of homogeneous and non-homogeneous media as the volume map decouples the representation of the radiance from the geometry of the media.

Building the volumetric photon map is different to that of the surface volume map as the photons can interact anywhere within the definition of the volume, there is defined for the volumes a cumulative probability density function that defines the probability of an interaction at all points x within the volume.

$$F(x) = 1 - \tau(x_s, s) \tag{2.7}$$

where τ is the transmittance which is computed by ray marching.

$$L_r(x, \vec{\omega}) \approx \frac{1}{\sigma(x)} \sum_{p=1}^{n} f_r(x, \vec{\omega}'_p, \vec{\omega}) \frac{\Delta\Phi_p(x, \vec{\omega}'_p)}{\frac{3}{4}\pi r^3} \tag{2.8}$$

2.8.3 Sub Surface Scattering

Sub surface scattering is caused by a class of participating media where the light transport of the material can be approximated by a function whereby the reflection of light at a point x is reflected at another point x' This causes a softening of the appearance of the material a common example of this is the human skin where a large amount of the reflected light by the skin undergoes some amount of sub surface scattering. The reason that this case of participating media is separated from the general class is that their are specific properties of SSS that allow for the calculation of the light transport more efficiently. Subsurface scattering can be described by the bidirectional surface scattering function BDSSRF (**?**), this is a generalisation of the BDRF and is given as:

$$S(x_i, \vec{\omega}_i; x_o, \vec{\omega}_o) = \frac{dL_o(x_o, \vec{\omega}_o)}{d\Phi_i(x_i, \vec{\omega}_i)} \tag{2.9}$$

From Equation 2.9 it can be seen that their are two points that needed to form the BDSSRF, the incidence location x and the reflectance location x', this can be seen as the light incident to x being scattered within the surface of the material and being reflected at the point x'. The BDRF can be seen as a special case of this function where their is the assumption that the location of emitted radiance is the same as that of incidence irradiance.

2.9 Photon mapping advances

Since the initial creation of the photon mapping algorithm in 1996 there have been many improvements and modifications to the photon mapping algorithm in order to increase the efficiency and the range of effects that can be visualised, in this section we shall summarise this work.

2.9.1 Reverse Photon Mapping

In order to speed up the photon mapping algorithm Havran et al. developed the reverse photon mapping algorithm, it is claimed in his paper that substantial speedups are possible by performing the ray-tracing stage before the photon mapping stage by accessing memory in a more coherent manner.

2.9.2 Progressive Photon Mapping

Progressive Photon Mapping (**?**) is a reformulation of the photon mapping algorithm into a multipass solution to the rendering equation in which the ray tracing stage is performed prior to the photon mapping stage in a similar way to reverse photon mapping.

In the traditional photon mapping algorithm the number of photons that are emitted into the scene are fixed, as such it is not possible to obtain an estimate of the radiance to an arbitrary precision, this can cause the estimate of the radiance to be blurred. Progressive photon mapping attempts to solve this problem by performing more than one photon mapping pass. After each photon map stage the search radius at each pixel is reduced, this allows details to be refined through subsequent passes. In addition to the ability to refine the image quality the memory requirement to produce an image to a given precision is reduced as each photon map does not need to be stored in main memory.

While this algorithm has been shown to be a useful addition to the traditional photon mapping algorithm it is much more complicated, we need to perform more than one photon mapping stage which require the radiance estimate to be recalculated multiple times.

2.9.3 Stochastic PPM

Stochastic Progressive Photon Mapping (SPPM) (**?**) is a recently extension to PPM that is similar in motivations as distributed ray-tracing, that is to be able to use the algorithm to produce phenomena such as depth of field, motion blur and glossy reflections. This method has been shown to converge to the correct solution for these phenomena and specular-diffuse-specular paths faster that normal PPM.

2.9.4 Dynamic Scenes

Although the photon mapping algorithm has traditionally been used for static scenes there has recently been work towards modifying the algorithm so that it can be used for dynamic scenes, the work of Weiss et al. (**?**) in this paper it presents a method of rendering that uses information from previous frames of the scene in order to gain a speedup of the rendering process.

2.10 Acceleration Structures

In order to produce an image of a non-trivial scene accelerating structures are a necessity, the amount of time to check intersections for each ray grows with order $O(N^2)$ if each object is checked for an intersection. Some candidates for structures are, Octrees, KD Trees, BSP trees. All of these data structures are designed to decrease the amount of time spent checking for intersections. It may be worth an investigation the performance characteristics of the various data-structures within the system, this can easily be done by making the underlying implementation of the data-structure transparent to the rest of the system.

2.10.1 K-D Trees

A kd-tree is a data structure that partitions space along split-planes which are axis aligned hyper-planes first proposed by Bently (**?**). The use of kd-trees in rendering applications is common as they reduce the time required to find the intersection of a ray and an object within a kd-tree. kd-trees are also needed within the photon mapping algorithm for the radiance estimate as we need to perform a neighbour search, Jensen uses this data structure within his implementation of the photon mapping algorithm in his book (**?**).

Building K-D Trees

Wald describes an algorithm for building a kd-tree with $O(Nlog(N))$ complexity (**?**), as stated in his paper the desire for more and more complex scenes requires that attention should be make to the building of the kd-tree as it can begin to take substantial amount of computing resources to perform. This paper is a good overview of techniques for building a kd-tree.

Surface Area Heuristic

The Surface Area Heuristic (SAH) is a technique often used in the building of kd-trees in order to decide upon the location of the split plane. First presented by MacDonald et al. (**?**) the SAH estimates the cost of splitting at a given point on a kd-tree, this is done by

estimating the cost of traversing the structure and the probability of a ray intersecting the objects within the structure. The SAH does make some assumptions such as the cost of traversal etc, as noted in (**?**) some of these assumptions are not strictly correct but the result of using the SAH are still valid.

2.10.2 Nearest Neighbour search

In order to produce a radiance estimate for a location on a surface we need to be able to query to photon map in order to find the nearest n photons, the kd-tree is a good candidate for this search as it is possible to perform the search with $O(log(N))$ complexity. The algorithm was first proposed in the paper introducing kd-trees by Bently (**?**) although as noted by Bently in later versions of the paper a more clear explination of the algorithm is stated by Freidman et al. (**?**).

2.10.3 Ray KD-Tree Traversal

Devoting time to creating efficient traversal algorithms is vital for all applications that include ray-tracing as a large proportion of the running time will be used performing these intersection tests, Fussell (**?**) describes an algorithm to perform this traversal that performs front to back search of the enclosing volumes allowing the closest intersection to be found efficiently.

2.10.4 Irradiance Caching

Along with the photon map, irradiance caching is a strategy to reduce the computation in order to produce a correct image, developed by Ward et al. (**?**). The algorithm stored irradiance values calculated on lambertian diffuse surfaces in a data structure that allows for the values to be interpolated for other points on the surface, this is done with the assumption that the values will be slowly varying over the surface. A thorough explanation of irradiance caching with respect to photon mapping can be found by Jensen (**?**)

Chapter 3

Requirements and Specification

In order to successfully develop the system for this project it is necessary to produce a specification for the system that will provide the constraints on the scope on the system that can be used to assess the progress of the product With this system the requirements are mainly motivated by providing a practical implementation of photon mapping, as a result this will be the most important requirement, as with any computer graphics product the requirements should be influenced by the desired result of the system, we can consider the ability of the system to simulate certain phenomena as being requirements, these phenomena, in particular we will concentrate on phenomena which photon mapping has been identified as being a good candidate for producing, this includes diffuse intereflection, colour bleeding, caustics and multiple scattering in participating media. Other requirements arise from more practical considerations, namely that the system should be able to run on a variety of hardware with different capabilities, in particular advances in CPU design has lead to multiple cores being common place allowing for parallel operation of our system as such we should attempt to design the system such that it scales well with respect to CPU cores.

3.1 Requirements

In this section we shall provide a listing of the requirements that have been decided upon for this project, these requirements will be used to inform the decision in the next chapter and to evaluate the success of the project. The terms must indicate that the requirement is not optional and must be implemented to fulfil the aims of the project, should indicates that the requirement is a feature that is desirable but may be forgone if constraints during implementation require it.

3.1.1 Functional Requirements

1.1 *Must Support Image Output.*
The program is designed to synthesis images, as such it is important to be able to save the images produced by the program.

1.2 *Must Support GUI output.*
It is desirable when creating an image to be able to see the results of the render as they are created.

1.3 *Scene input must be in human-readable format*
This is so as editing scene files is possible without the use of other programs.

1.4 *Must accept all scenes in valid format*

1.5 *Must not accept invalid scene files*
The user should be informed of invalid input.

1.6 *Must perform photon mapping algorithm*
As this is the focus of the project it is a key requirement of the final product.

1.7 *Must support diffuse and specular reflectance properties*

1.8 *Must support generic scene objects*

3.1.2 Non Functional Requirements

2.1 *Must Support Linux and Windows*
Cross platform operation of the program must be possible.

2.2 *Should Support Mac OSX*
Due to hardware availability this is less strong of a requirement than 2.1

2.3 *Should support user configuration from the command line*
Users should be able to configure the render parameters without recompiling.

2.4 *Should perform synthesis in a reasonable time*
The use of algorithms and acceleration structures that lower rendering times should be used where possible.

2.5 *GUI should be responsive to user interaction*

2.6 *Results must be consistent across runs*
The program should not demonstrate different behaviour for different runs with the same input data.

2.7 *Scene input format should be documented.*
Users should not be expected to read the source code in order to create a scene for the program.

2.8 *Software design must be modular with clear separation of concerns*

Chapter 4

Design

In this chapter we will discuss the design aspect of the project, this will include an overview of the architecture that we have decided upon and the rational behind it, we will then move to describe key components of the design and how they relate to the aims and goals of this project and any interesting problems that they may cause in the implementation and how we propose to solve the problem.

4.1 Architectural Overview

During the requirements analysis stage of this project it was clear that the project could be decomposed into two main sub-modules, a front-end that will perform the interaction with the user and the operating system and a back end that will process the user input in order to synthesis image, this can be seen in Figure 4.1, further decomposition of the back end was decided from the definition of the photon mapping algorithm which is conceptually separated into two stages, the photon emission stage and the ray-tracing stage, it would be natural then to subdivide the back end into separate blocks that perform these steps, the first producing the photon maps and the second using this data along with the scene description from the front end to produce the output image.

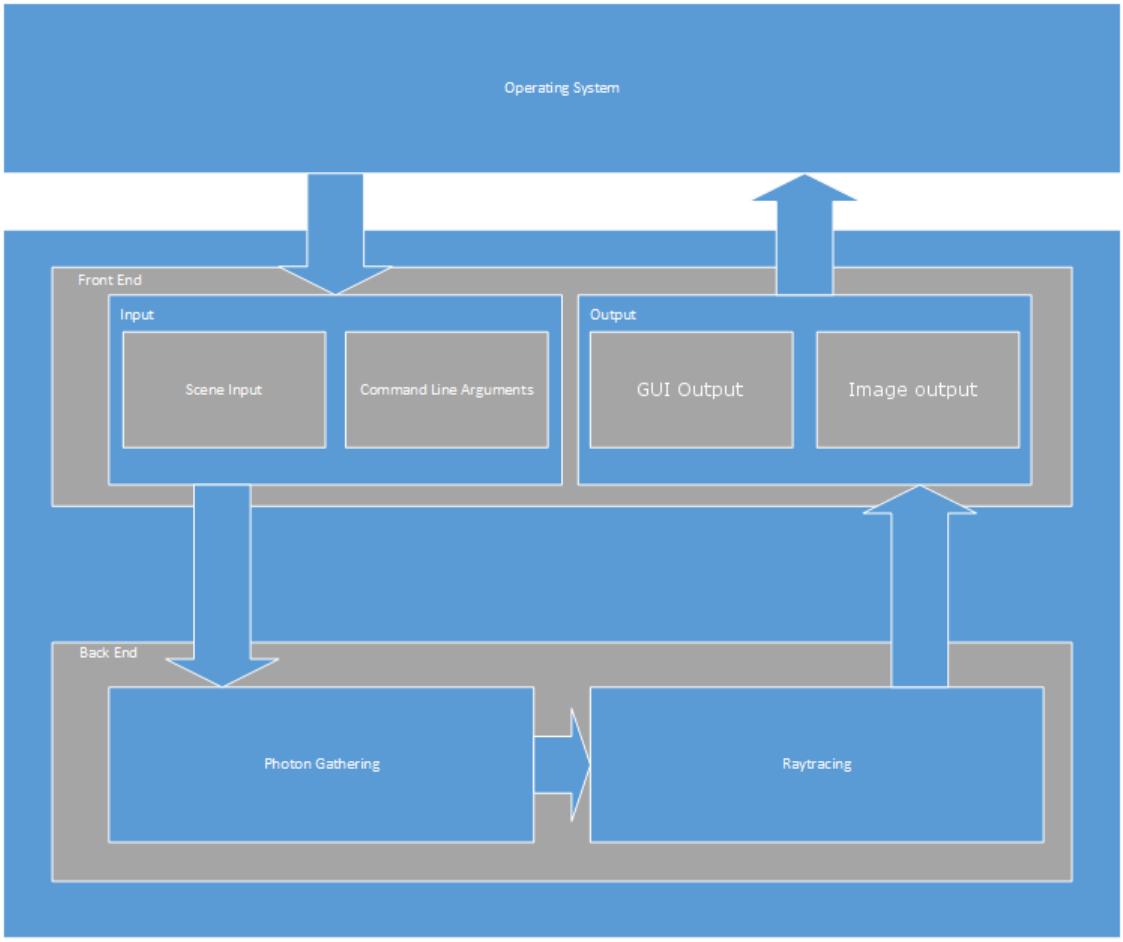


Figure 4.1: Top level block diagram of the system

4.2 Front End

The front end of the system deals with the input and output of the system, this will manipulate and process the input into a form that can be used by the back-end to perform the synthesis, separating the front and back-end allows the i/o and the synthesis to be modified with no modification required for the other block, for instance the output could be an BMP image or a surface within a GUI, the details of the conversion are irrelevant to the back-end, only that the data passed to it is in a format that is understood.

4.2.1 Scene Input

The scene input takes in a description of a scene that is to be rendered from a file, this scene description will give a listing of each of the objects in the scene such as meshes, lights and camera. Each of these objects are also described by a file that is listed in the scene file. Each of the scene files are text files that are human readable, this allows the files to be easily modified. Each of these configuration files allows for attributes to be reused such as a material describing a mirrored surface can be defined in a single material file that can be referenced by multiple objects. below is a list of the different scene files that are defined

.scene Top level description of the scene

.mesh Description of a mesh, including material and surface data.

.mat Description of a objects surface properties.

PLY mesh format

One aspect of the mesh input file is a file-name that refers to a PLY polygonal mesh, it was decided that using a pre-existing file format for the input to system was beneficial as it allowed for assets from 3D modelling utilities to be used to create the test scenes as they contain the required functionality to export to PLY meshes. the choice of the PLY made because it is a human-readable format (a binary PLY format is available but not supported by the system) and we specify this to be a key requirement of the system, using PLY also allowed for example meshed from the Stanford PLY repository which in turn provided a selection of relatively complex models that could be used during the production and testing.

4.2.2 Command Line Arguments

The program that is created must be able to take in more than one scene file to be useful, by having command line arguments this allows the scene file to be changed without recompiling the code, there are other options that are available including the number of threads that

../Code/data/scenes/cornell_box.scene

```
mesh ./data/mesh/cornell_box_box_main.mesh
mesh ./data/mesh/cornell_box_box_left.mesh
mesh ./data/mesh/cornell_box_box_right.mesh
mesh ./data/mesh/water.mesh
sphere 0.30 -0.5 -0.70 -0.3 ./data/materials/mirror.mat
sphere 0.30 0.5 -0.7 0 ./data/materials/glass.mat
camera 0 0 5 0 0 0 0 1 0 4
light point 0 0.999 0 30 30 30
#light area 0 0.99 0 0 -1 0 7 7 7 2 2
```

Figure 4.2: Example Scene File

will be used to perform the ray-tracing and the resolution and name of the output file, this will allow for the system to be used in scripts.

4.2.3 Image Output

The image and GUI block contains the code that will take the final pixel colours and present them to the screen there will be two modes of operation, image output and GUI output, image output saves the image to a file such as png or bmp.

4.2.4 GUI

In order to provide immediate feedback to the user it is necessary to include a GUI in the system, this will be a simple window that will display the progress of the system as it creates the images, this is useful for testing changes to a scene as it may be possible to see if the change has had the desired effect without finishing what may be a costly render or to assure the user that a render is still running, the block level design of the GUI can be found in Figure 4.3.

4.2.5 Console Output and Logging

In certain environments it is not possible to support a GUI (headless server) as a result the system includes output on the console that inform the user of the progress of the render, this provides a visual display of the progress at each stage in the rendering and also any warnings that can be used to debug issues that arise during the execution of the program.

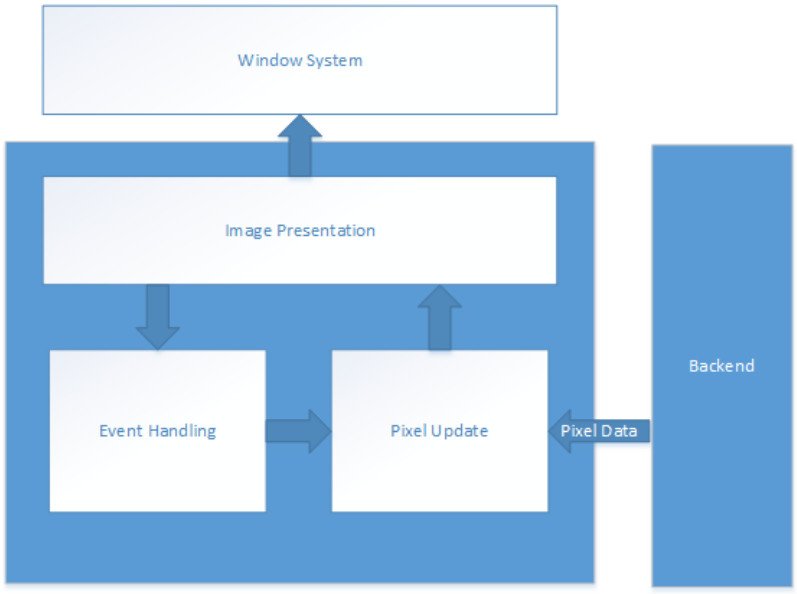


Figure 4.3: GUI design diagram

4.3 Back End

The back-end contains the code that deals with processing scene data, this block is decomposed into two main blocks, the photon mapping block and the ray-tracing block, the photon mapping block is responsible for the prepossessing step that creates the photon maps used to estimate the radiance. The ray-tracing stage is responsible for computing the colour of each of the pixels in the output image, this will use the data that was read into from the front end and the photon maps that were created by the photon map module. The output of the ray-tracing block is sent to the front end which will use the data to present or save the final image. Both the photon map generation and ray-tracing stages have been designed as pipelines with data being passed between threads in a producer consumer pattern through thread-safe queues.

4.3.1 Photon Emission and Photon Map Construction

We further decompose the photon map generation into two stages, the photon emission stage which traces photons through the scene and stores the point of absorption and the balancing stage which sorts the photons into a data structure (left balanced tree) that facilitated efficient storage and searching. The reason for the balancing stage is that the distribution of the photons in the map is unlikely to be optimal (**?**).

The construction of the photon map can be computationally expensive when a high number of photons are requested to be created, in order to improve the performance of this stage multiple threads are created that are responsible for emitting a proportion of the photons in

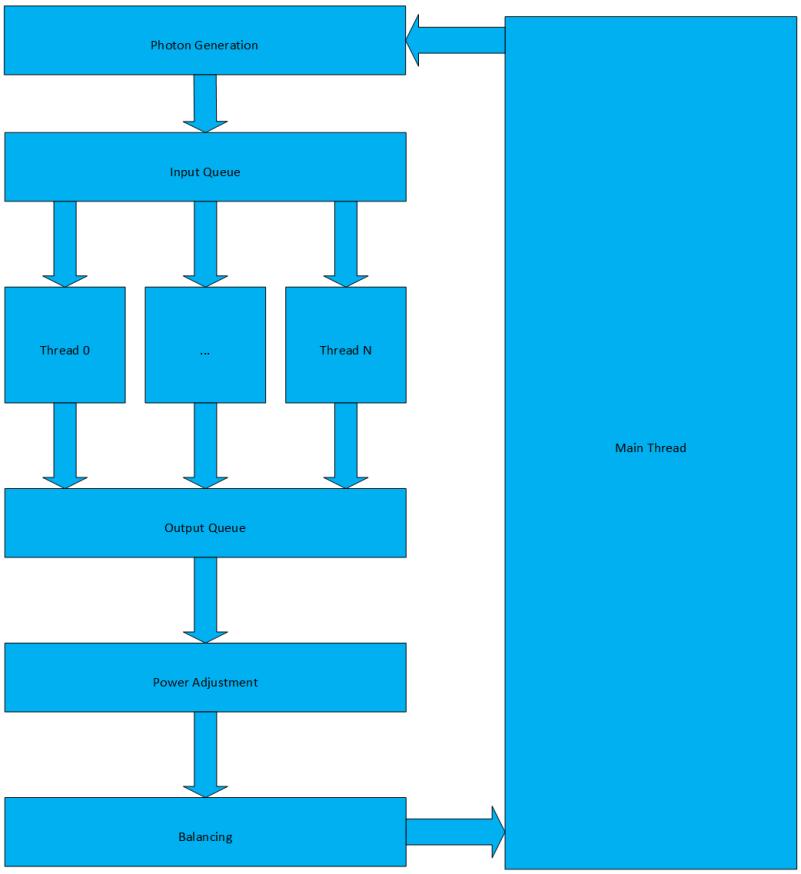


Figure 4.4: Photon map generation

the final photon map, the number of threads will be set in the front end and is configurable by the user to suit the machine that the system is being run on.

Each of the threads will trace a photon through the scene, at each diffuse interaction it will write the photon to the output queue, the photon will then be reflected to continue the photon map construction.

The next stage in the pipeline reads photons from the queue, this will continue until each of the threads has signalled that they have completed processing the photons, once all of the threads have been completed the the list of photons will be transferred into the photon map as a left balanced tree.

Multiple Photon Maps

During the construction of the photon maps a choice must be made as to which of the photon maps the photon should be placed in, this requires additional information to be passed from the emission thread, this information will describe the path that the photon

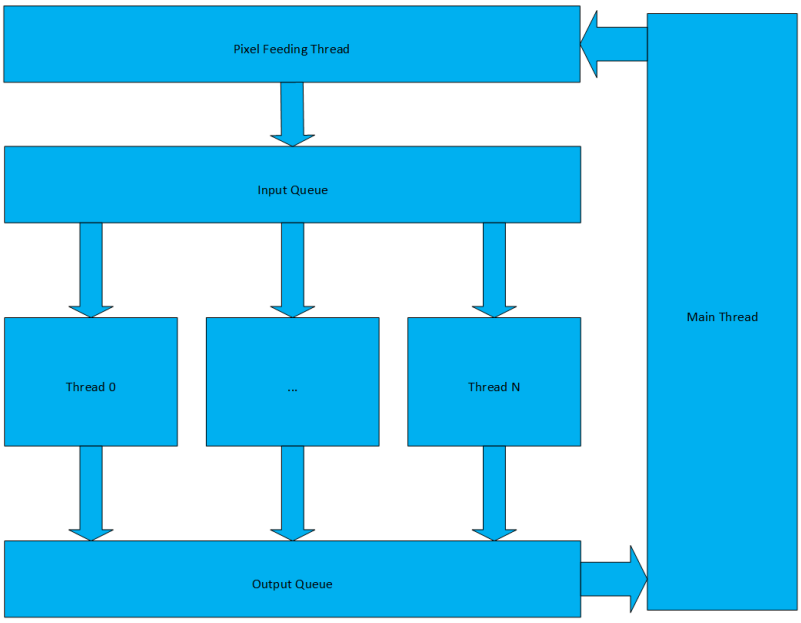


Figure 4.5: Ray-tracing architecture

has taken, this will in essence be an implementation of the path notation that is commonly used to describe the path of a photon, in this case all paths will be stored in the global photon map ($\mathbf{L(S|D)^{+}D}$ in the path notation),only specular paths are stored in the caustic photon map as these are the paths that contribute the most to the focusing effect of caustics ($\mathbf{LS^{+}D}$) note that we have no eye component as the photons are only traced to diffuse surfaces.

4.3.2 Ray-tracing

As can be seen from Figure 4.1 the ray-tracing module contains multiple threads, each of these threads trace a subset of the pixels in the image and run independently to each other, this is possible due to the inherent parallelism in the photon mapping algorithm. This stage in the system will perform ray-tracing as described by Shirley (**?**), for each intersection found during the running of the system the photon maps created during previous stage will be used to estimate the radiance of the point in order to estimate the true illumination at the point factoring in the global effects.

The architecture of the ray-tracing block shared many similarities with that of the photon emission stage, it is also a pipeline, there is a threads that inputs each pixel in the output image to a queue, this queue is in turn read by one of multiple threads that perform photon mapping for a ray emitted through the pixel, multiple rays are generated for each pixel performing distribution ray-tracing, the colour of the pixel is then written to an output queue that will be read by the main thread in order to send the pixel values to the front

end to be presented to the user.

4.3.3 Objects

Much of the ray-tracing algorithm that is performed on each of the objects in a scene is the same, for example finding a refracted ray is only dependant on the surface normal at a point of intersection for this reason the system abstracts any object that can be intersected must implement certain functions, this is a form of object orientation.

Materials

In order to determine the path of the photons that are emitted and determine the colour of the pixels when performing ray-tracing we require a description of the reflectance of the surface, we have decided to model perfect diffuse surfaces and perfect specular reflectance and transmission, we store this information as coefficients $\sigma_d, \sigma_r, \sigma_t$, each of which contain a red, blue and green components, participating media also require two more coefficients, the absorption and scattering coefficient again with three colour components.

4.3.4 Shading

Each thread is responsible for performing that shading at the intersection of eye rays, for specular reflection and transmission this results in an additional ray-trace be performed to determine the shading colour, for diffuse surfaces a direct illumination calculation is performed, additionally the photon map is queried to determine the indirect lighting. For participating media, the radiance is calculated by performing by a ray-marching procedure.

Chapter 5

Implementation

In this chapter we will discuss the main implement ion issues that arrised during the development of the project and the approach that we took to implement the design from the previous chapter, we will begin by describing the software tools that were used throughout the implementation, then we will discuss the implementation of the front-end , object system and finally the back-end of the system.

5.1 Tools

5.1.1 Implementation Language

The implementation language that we have chosen is C, specifically C11. The choice of C is because it it the language that I am most familiar with, also it is a language that offers performance benefits as it is a compiled language, this is of great advantage for a photon mapper as we will regularly be dealing with data structures that contain thousands to millions of elements (such as the photon map.) In addition to being a compiled language the memory management of the program is left to the programmer, while this can cause issues such as buffer overflows it has the benefit of providing a great deal of control to the program writer as they can be aware of all memory that is used by the program, which, when dealing with data structures as large as the photon map is a large benefit.

5.1.2 Coding Style and Code Quality

With any piece of software it is important to impose restrictions on the way in which the code is written in order to reduce the effort that is needed to read the source. Some of conventions that we have used includes having a convention for any typedef such that a type can be easily determined (appending _t to the end of the typename), requiring that if a function is non-static that it begin with the name of the file or another prefix to prevent

Figure 5.1: Example Valgrind Output

name clashing. Another convention that is used is the use of opaque pointers (**?**) to data structures in order to hide the internal data representation of the types an example of this can be found in the source files `list.h/c` it can be seen that the header file does not expose the structure definition to any file that includes the header.

Throughout the development of the system it was necessary to use a number of tools in order to ensure the quality of the code being produced.

Another issue that is present in writing code for C is that the memory is managed by the programmer, as a result it is common for errors such as memory leaks to occur, in order to reduce the risk of this in the project we have utilised several programs that can identify issues.

5.1.3 Valgrind

In order to detect memory related bugs we used Valgrind which is a program that will track all memory allocations of a running program and is able to detect errors such as writes after a region of memory that has been allocated and lost pointers that result in memory leaks.

5.1.4 Static Analysis

Included with the clang compiler suite is the clang static analyser, this program will perform analysis on the source code of a program and will report certain types of errors such as performing memory accesses on variables that can potentially be NULL, while this may seem similar to valgrind, this approach will detect errors such as this without running the program and can find issues that may only manifest in certain corner cases that may not appear during testing.

5.1.5 Source Control

In order to organise the code that we have developed we decided to use Git as a source control system for the project. Git has the concept of branches that allow development of separate features concurrently in isolation and merged upon completion, incorporating branched into the development work flow proved to be a highly useful feature as it allows for work on a feature to continue even if another feature breaks parts of the system that would otherwise make continued development impossible

5.1.6 Profiling

When writing software it is desirable to optimise it in order to improve the performance of the code, this project is no different and there are many area that could be optimised, to quote Knuth

> Premature optimisation is the root of all evil ... in programming (**?**)

This refers to the practice of optimising code that is not a bottleneck in the execution of the program, for instance reducing the running time of a function by 50% that is only executed for a small proportion of the running time of the whole program in order to decide which parts of the code needs to be optimised we have used the combination of the compiler profiling flag and gprof, a program that will output profiling statistics for a run of the executable, this includes information such as the number of times a given function was called and the percentage of the run time taken by the function.

5.2 Common Data Structures

In this section I will briefly discuss some data structures that are used as part of the system that cannot be categorised as being in the front or the back end.

Due to the small standard library provided by C many features that are commonly found in more high level languages are not found in C, for this project we need three such data structures, lists queues and vectors.

5.2.1 Lists

As we don't know how many objects and lights are in the scene and number of photons stored in the photon maps depends on the lights and geometry in the scene, as a result we have implemented a list data structure that allows for the size of the list to increase as elements are added to the list. The implementation of this data structure is quite simple, a list stores a pointer to an array in memory, the size of the elements in the array, the number of elements in the array and the capacity of the array. If an item is added to the list and the capacity is two low the capacity is increaed by calling the function `realloc` allowing for dynamic memory managment to be encapsulated.

5.2.2 Queue

During the implementation of the system we frequently required to coordinate between multiple threads for example during photon emission and ray-tracing, the solution that we decided upon in both cases was to use a thread-safe queue implementation, this is implemented as a fixed size circular buffer, two pointers are stored to the front and back of the queue, as items are read from the queue to front of the queue is decreased and writing to the queue increases the back pointer.

5.2.3 Vectors

Some of the most common operations that are performed during photon mapping and ray-tracing are vector operations. We provide a simple implementation of vectors in this system as facilitate operations such as vector addition, subtraction, dot and cross product and so on. The implementation of these operations are not particularly optimised although we were able to observe that our implementation was optimised by the compiler to utilise vectorisation instructions available for the architecture that we have tested the system on.

5.3 Front End

As noted during the discussion of the design of the system the front end of the system is responsible for the processing the input from the user and relaying the data produced by the back-end back to the user.

As this stage is primarily responsible for performing input output operations it is unlikely to end up being a bottleneck in the operation of the system, as a result the implementation has not been optimised to perform its role as efficiently as possible to allow more resources to be allocated to the development of the back-end.

5.3.1 Scene Input

Scene input begins by reading from a file pointed to by the user and if not given a default file defined in the source code, this file is a list of all objects in the scene each object described is created and stored in structure (see Figure **??**) each of the objects in the file may be completely specified from the scene file such as lights and cameras while other objects such as meshes and spheres may require other files to be read containing data such as material properties and vertex information.

Mesh Input

If a mesh is specified in the scene file the vertex information will be read from a ply file, this data is just a list of vertices, normals texture coordinated and triangle indices, this structure is not conducive to efficient intersection tests, as a result a k-d tree is created before storing the mesh in the scene, k-d tree construction will be discussed in Section 5.4.2

5.3.2 Command Line

Users are able to input command line arguments to the system, these are received by C programs in the input parameters of the main function, these input parameters need to be parsed and stored, this is performed by simply scanning over each option in the argv list to find a valid option or pair of options in the case of a user inputted value for the option i.e. width of the output image.

```
./raytracer -w 1000 -h 1000 -i ./data/scenes/cornell_box.scene
```

Figure 5.2: Example command line.

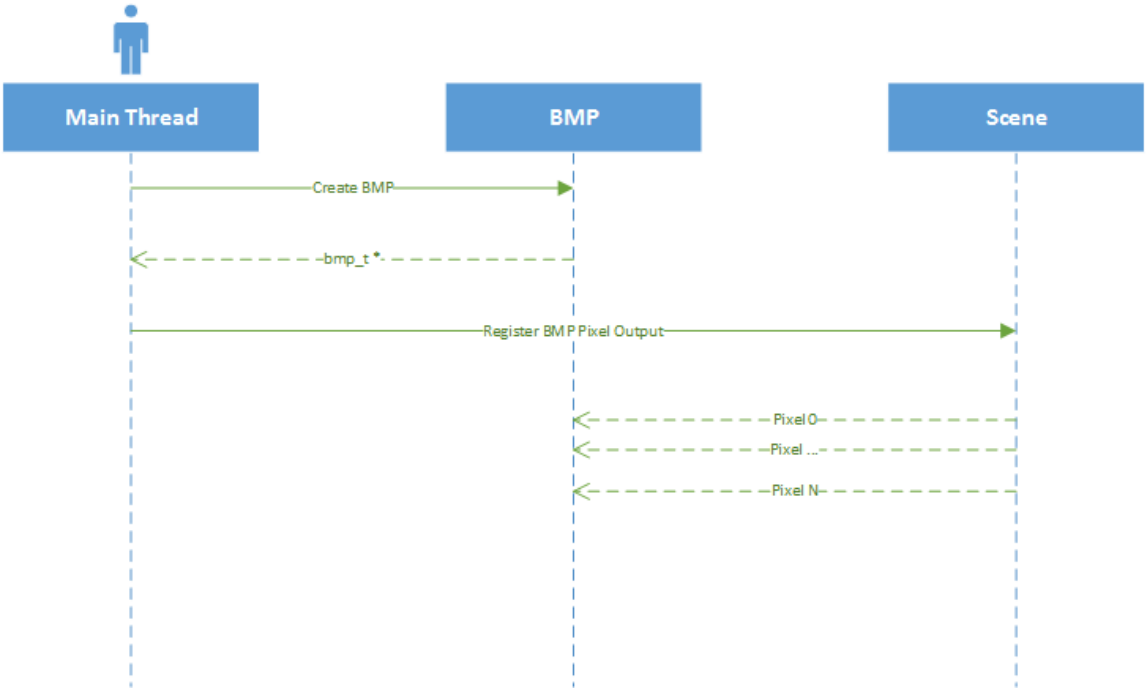


Figure 5.3: Sequence for pixel update

5.3.3 Global Configuration

Once the configuration of the scene has been read and processed the data containing the configuration will not change for the lifetime of the program, as a result the global configuration is available in the system through a global variable, while it is generally advised when developing software that global variable should be avoided an implementation that passed the configuration would require that a pointer to a configuration structure be passed, this amounts to having the same result as a global variable whilst reducing the clarity of the code, although we do lose the ability to specify a const qualifier to prevent modification and as a result care had to be taken that no variable in the configuration structure was modified.

5.3.4 Common Pixel Update Interface

As part of the decoupling of the input/output of the system and the image synthesis we have decided to create an interface between the from and back-end that will be used to send pixel data to the front end, this interface allows for a callback to be registered that will be called for each of the pixels in the output image, multiple callbacks can be registered allowing for the interface to be used for multiple purposes, Two modules have been created that use the interface to demonstrate the usage of the interface, these are described below.

Image Output

The systems output default image format is BMP, this image format was chosen as it is a simple image format, there are several variants of BMP each with its own header definition the version of BMP that is implemented is the OS/2 bitmap this is one of the more simple variants with a fixed size header that specified few options. The implementation of the BMP output can be found in `bmp.h/c`. If supported on the platform the system is also capable of outputting images in PNG file format, this is achieved by using libPNG, there are advantages to outputting to this format, foremost is that the image size of an image output to PNG is significantly smaller than that outputted by BMP.

GUI

The user interface that is included in the system is designed to allow instant feedback to the user of the render, this can be useful in the case of errors that can be identified early in what may be a costly render, the interface is rather simple providing a surface that displays the current state of the render.

We have decided to use SDL library to provide the interface to the operating system window system and OpenGL to perform the drawing of the pixels to the screen. SDL (Simple Direct media Library) is a library that allows for cross platform applications to be written that include drawable surfaces, OpenGL support and user input, SDL is written in C and provides a library API for C.

The surface that the image is displayed on is implemented as a OpenGL textured quad that spans the width and height of the screen. The callback function that is defined for the GUI writes each pixel value to a queue which will be read by the render mainloop, this mainloop will read the pixel data and update the appropriate location in the texture with that value, in order to reduce the number of draw calls made by the GUI multiple pixels can be updated before each draw call, this is implemented by performing a non blocking read until we cannot read any pixels without blocking, we will then perform the draw call.

In order to retain responsiveness of the GUI while rendering it is desirable to run the GUI on a separate thread, the GUI should also only redraw the screen when a pixel has updated its colour, this is to reduce the amount of processing that the GUI is performing that could be used to create images, on the other had the GUI should respond to events that are pushed to its internal event queue such as key presses and window system events, Figure 5.4 demonstrates the GUI running.

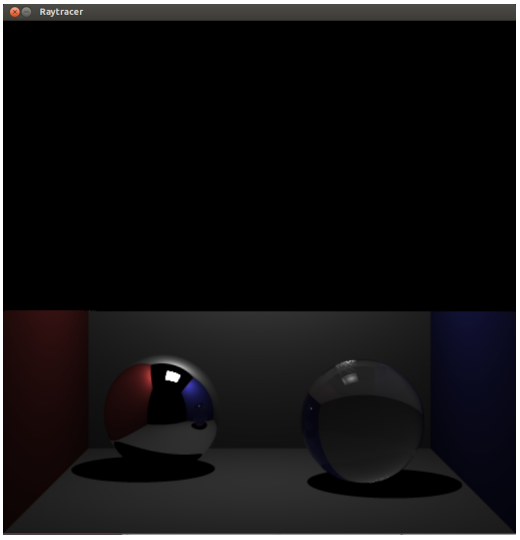


Figure 5.4: Screenshot of render in progress

5.4 Object System

As discussed in the requirements and design sections we include in the system a generic description of the objects in the scene , this is implemented as a structure that contains function pointers that perform functions that must be supported by all raytracable objects and the data related to the individual object. By encapsulating the data and the functions performed by the object it is possible to create algorithms on objects in the abstract such that the details of the object can be ignored, for instance calculating a reflected ray at a point if intersection is a function of only the surface normal at that point, whether it is a triangle mesh or sphere the same ray will be calculated, a full list of the functions that need to be defines is given in Figure 5.5

We will now attempt to give an overview of the implementation of the objects currently used within the system.

5.4.1 Sphere

The sphere primitive is defined implicitly as an origin point and distance from the origin, including a sphere primitive is a natural choice as many of the operations that we need to perform such as intersection tests and finding the normal at the point of intersection is much simpler for spheres that most other surfaces allowing us to use this primitive for testing changes while being confident that any errors that we find are not within the object definition of the sphere, the same cannot be said for a mesh where the optimised intersection test used is fairly complex and contains more scope for implementation errors.

int (*intersection_func)(object_t *o, ray_t *ray, intersection_t *info)
Tests if the input ray intersectes with the object, returns the result and any other information is stored in the input variable info

void (*bounds_func)(object_t *o, aabb_t *bounds)
Returns the bounds of the object which is used during the scene-ray intersection test.

void (*normal_func)(object_t *o, intersection_t *info, double *normal)
Returns the normal of an object at a given intersection point defined in info.

void (*tex_func)(object_t *o, intersection_t *info, double *tex)
Returns 2-d texture coordinates in tex of the object at the point of intersection info.

void (*delete_func)(object_t *o)
Frees any memory and other resources used by the object.

void (*shade_func)(object_t *object, scene_t *scene, intersection_t *info)
Returns the colour at the surface of the object at the intersection point defined in info the result is stored in info.

Figure 5.5: Functional Definition of an Object

Intersection test

Given that we define a ray and a sphere in a parametric form it is intuitive to solve the intersection of these objects implicitly, given a ray defined $\vec{o} + t\vec{d}$ and the equation for a sphere $(p - c) \cdot (p - c) = r^2$ where p is a point on the the sphere, c the centre of the sphere and r the radius of the sphere, combining these two equations we get:

$$(\vec{o} + t\vec{d} - \vec{c}) \cdot (\vec{o} + t\vec{d} - \vec{c}) = r^2$$

expanding this gives us:

$$(\vec{d} \cdot \vec{d})t^2 + 2(\vec{o} - \vec{c}) \cdot \vec{dt} + (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2 = 0$$

this can be solved as a quadratic equation of the form $At^2 + Bt + C = 0$ where:

$$\begin{aligned} A &= \vec{d} \cdot \vec{d} \\ B &= 2(\vec{o} - \vec{c}) \cdot \vec{d} \\ C &= (\vec{o} - \vec{c}) \cdot (\vec{o} - \vec{c}) - r^2 \end{aligned}$$

as this is a quadratic equation we have two possible solutions for t given by:

$$t_0 = \frac{-B - \sqrt{B^2 - 4AC}}{2A}, t_1 = \frac{-B + \sqrt{B^2 - 4AC}}{2A}$$

for the case when $B^2 - 4AC < 0$ we have no real solutions and the ray does not intersect with the sphere, otherwise we take the smallest positive intersection value, if both values are negative we do not intersect the ray.

5.4.2 Mesh

The mesh primitive is defined as a list of vertices and indices that define the triangles from the list of vertices, also stored are the surface normals for the triangles and if included the texture coordinates. We will not discuss the implementation for reading in the mesh in detail as it is a rather simple function that reads from a PLY file and extracts the data into the form as described during the discussion of the system design.

K-D Tree Construction

Intersecting a triangle mesh with a brute force method would be prohibitively expensive for a mesh with many triangles, it is common then, to use a acceleration structure that reduces the number of intersection. The construction of the kdtree is performed during the initialisation of the triangle meshes as it is a static data structure. We begin the construction of the kd-tree with all triangles in a single list, we then calculate an axis and position on that axis to separate the triangles, this is calculated by finding the median spread of the geometry in the voxel we then sort each of the triangles into two child lists, if a triangle is to the left of the splitting plane it will be placed in the left child's triangle list and visa-versa for the right, if the triangle spans the split plane it will be added to both lists, we then call the tree building function recursively on the left and right child and delete the list for this node, we terminate when a list contains less than a threshold number of triangles in its bucket or a maximum depth is reached or if all triangles are placed in a single node.

Intersection test

Given the k-d tree that we have constructed for a mesh we can now describe an efficient top-down intersection algorithm, we begin at the root of the k-d tree. Given that the tree has been split along an axis we can calculate which of the child voxels are closer to the ray by calculating the values of intersection for the two voxel, if the ray origin is within a voxel it is set to be the near voxel and the other child the far, otherwise we compare the t values for the voxel intersection. Once we know which voxel to traverse, if the ray exits

the near voxel before crossing the splitting plane we do not need to test the far voxel as we cannot intersect any geometry in that voxel, we perform this procedure recursively on the near and possibly far voxel until we reach a leaf node, we then test all triangles in the index list, if we find an intersection we record the t parameter for the intersection and check the rest of the list, if an intersection if found we can return the intersection point and discard any other voxels that we were going to check as they cannot have an intersection closer. There is a subtle point that must be noted, when we are checking for that intersection of a triangle in a list if the intersection point is found to be more that the exit point of the voxel then we do not record this intersection as the triangle spans voxels and we need to check for closer intersection.

5.4.3 Material Properties

Each object has associated with it a material property that defines the reflectance properties of the material. The system currently supports diffuse and pure specular components each being defined as a three floating point values for red, green and blue components respectively, texture mapping is also supported for the diffuse component, for specular transmission an index of refraction is defined that determines the direction of refracted rays through the material. We also store the average values for these components explicitly as they are frequently used when performing Russian roulette to evaluate integrals during photon map construction and ray-tracing. Participating media require additional components for the scattering and absorption coefficients as discussed in the Literature Survey, these are also stored as rgb components, we also store derived properties, namely the extinction coefficient and albedo of the ma trial.

We will now discuss the back end of the system, we begin with a discussion of the implementation of the photon map generation stage and move onto the ray-tracing stage and how we utilise the photon map in the global illumination calculations.

5.5 Photon Map Generation

We described in Chapter 4 that the photon generation stage of the back end would run on multiple threads to utilise the computing power of the machine the system is being run on, this required certain decisions to be made to allow for the multiple threads to correctly distribute the power of the light sources into the scene.

Each thread is responsible for producing a certain proportion of the photons in the scene, once a thread has processed all of these photons it will signal to the main thread that it has finished by sending a flag to the output queue the thread also updates a global light emission count include the photons that the thread emitted for each of the lights, a separate count for each of the counts is kept as the processing of the threads may not happen at the same time.

5.5.1 Photon Emission

In order to trace photons into the scene we first need a method of creating photons, this is implemented in the light definition, each light object has a associated function that will produce a random photon from the light that is consistent with the type of light, for instance a point light will generate a photon with origin exactly at the origin of the light source and in a random direction, an area light produces a photon with origin on the area of the light with direction taken from a cosine weighted hemisphere distribution in the direction of the normal of the light. Each photon that is emitted from the light sources begin with the full power of the light source, this will be scaled by the number of emitted photons from that light after all photons have been gathered.

5.5.2 Photon Tracing

Once we have created a photon from a light source we now need to trace the photon through the scene and record the interactions of the photon until it is absorbed or leaves the scene, the function that performs this function is `trace_photon` the declaration of this function is given below.

```
int trace_photon(scene_t *scene, ray_t *ray, int light, double power[3], bool specular, bool diffuse, bool specular_only;
```

The parameters ray, light and power defines the photon properties, ray defines the origin and direction of the photon, light is an index to the light from which this light was emitted

and the power contains the flux of the photon. specular and diffuse define describe the path that the photons have taken prior to the call to `trace_photon`

Photons are traced much like a ray in traditional ray-tracing, first we calculate the closest intersection point of the photon . Once an intersection is found the photon interacts with the surface of the object, this interaction can be one of specular reflection , transmission, diffuse reflection or absorption, at this point we must determine the path of the reflected photon, the power of this photon is determined by the reflectance of the surface i.e a photon with white light that interacts with a red surface will only reflect photons with power in the red component. Using the approach of scaling the photons will create a photon map that correctly describes the distribution of power in the scene, unfortunately this can lead to many photons with low power as they are scaled at each interaction, it can also in the case of surfaces with both specular and diffuse component lead to exponential number of photons being produced as we need to create a reflected photon for both paths scaled appropriately, as a result of these considerations we have used a monte-carlo method that reflects at most one photon per surface interaction with full power, we ensure the correct result in the photon map by reflecting the photon with probability based on the reflectance of the surface (i.e reflecting 50 full power photons as oppose to 100 half power photons) we sample the reflectance of the surface by forming a cumulative distribution function of the materials reflectance coefficients. Given the specular reflective, transmissive and diffuse reflectance coefficients ρ_s, ρ_t, ρ_d of the object we can create a cumulative distribution of photon emission paths (Figure 5.6) we then take a uniform variable ξ between 0 and 1 if the variable is within the distribution we reflect a photon, otherwise the path of the photon is terminated. As we are using rgb values to store reflectance properties the coefficients used to perform the sampling are the average of the three components, so as the distribution of the three colour bands is correct we must also perform scaling of the power parameter that we use in the next `trace_photon` call, note that the scaling preserves the power of the photon and the reflectance properties of the surface.

Figure 5.6: Russian roulette distribution

At each non-specular interaction the photons are written to the output queue to be processed, this includes the power of the photon, the incident angle of the photon to the surface and flags to indicate if the surface has interacted with a diffuse or specular surface prior to this interaction, we do not store photon interactions at specular surfaces as they do not provide us with information that can be used when estimating the radiance as the probability of an incoming photon contributing to the specular reflectance is zero due to the delta functions in the specular BRDF.

When creating the caustic photon map we are only concerned with those interactions with diffuse surfaces that have taken a path that contains at least one specular bounce and no

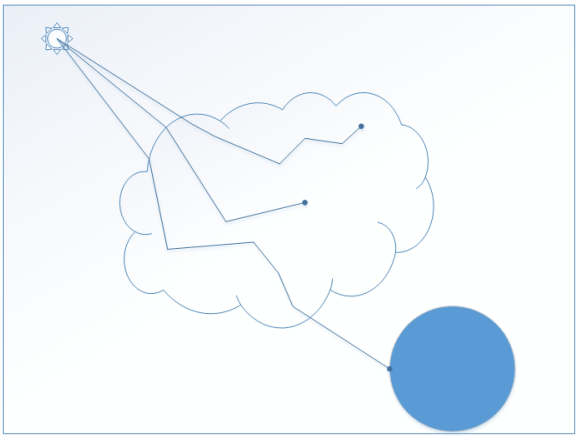


Figure 5.7: Random walk of photons through participating media

diffuse bounces (**LS$^+$DE**), as a result `trace_photon` has as input a flag that will indicate that any path that does not match this description should be discarded.

Participating Media

If the photon interacts with a participating medium the photon will begin a random walk inside the medium, the path of this walk is determined by the properties of the medium, these being the scattering and absorption coefficient, the sum of which is called the extinction coefficient, at each point in the random walk the photon is stored and then can either be scattered or absorbed, the probability of being scattered is determined by the Albedo Λ

$$\Lambda = \frac{\sigma_s}{\sigma_e} \tag{5.1}$$

Each step in the random walk continues for a distance until the next interaction with the medium occurs, the distance for the next probability if determined by the properties of the medium, the probability of photon travelling a distance Δx is given by $1 - e^{-\Delta x}$ (**?**), we can importance sample this distribution to determine the distance for the next interaction (**?**) which is given by:

$$\Delta x = \frac{-log(\xi)}{\sigma_t} \tag{5.2}$$

where ξ is a uniform random variable in the range [0, 1]. If a photon leaves a participating medium or intersects with an object in the medium it is treated as if it was not in the medium and is stored in the global photon map.

5.5.3 Photon Processing

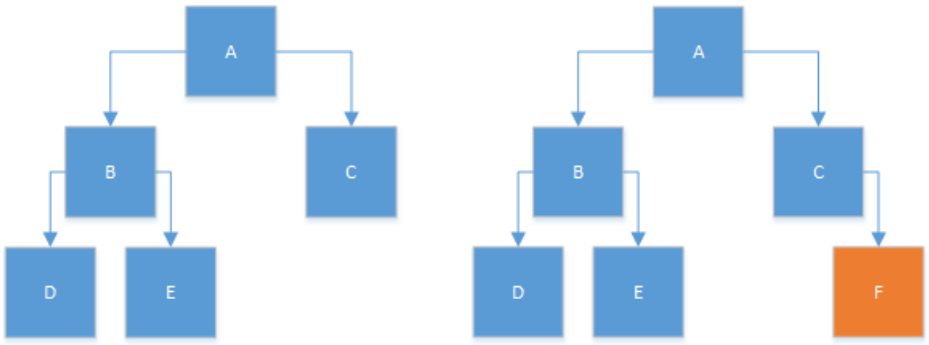
Each thread that traces the photons in the scene writes to a common queue, this queue is read by a single processing thread that classifies the photons by the path of the photon and stores the photons in one of three lists, the global, caustic and volume photon list, this thread will continue to read photons from the queue until each of the threads have signalled that they have completed their portion of the photons, this signal is in the form of a flag that is passed in the same queue as the photon data. After all photons have been collected the power of the photons are scaled by the number of photons that were emitted (note this is not the number of photons in the list but the total number emitted) from the same light as the photon.

K-D Tree Balancing

When performing radiance estimations with the photon map we will be performing nearest neighbour searches on points within the map, this requires the photon map to be arraigned in a manner such that this search can be performed efficiently. In order to do this we store the photon map in a left-balanced k-d tree. A left balanced tree is a tree structure where at each level of the tree the depth of the children differs by at most one, this allows us to store the photon map in an array with the location of the children in the photon map known implicitly, for a photon in position i the children of the photon can be found at the $(2i+1)^{th}$ and $(2i+2)^{th}$ location for the left and right tree respectively, the next stage is two transform the photon lists into a k-d tree.

Algorithm 1 Balanced K-D tree construction

```
function BALANCE(photons, index, max_index)
    if photons.size >1 then
        axis ← SELECT_AXIS(photons)
        left, median, right ← PARTITION_AROUND_MEDIAN(photons, axis)
        left_index ← 2 * index + 1
        right_index ← 2 * index + 2
        if left_index <max_index then
            left ← BALANCE(left, left_index,max_index)
        end if
        if right_index <max_index then
            right ← BALANCE(right, right_index, max_index)
        end if
      return left + median + right
    else
      return photons
    end if
end function
```



(a) (b)

Figure 5.8: Left Balanced Tree
(a) demonstrates a left balanced tree while (b) does not due to node F

Selection Statistic

In Algorithm 1 we partition the photons around the median of the list of photons, in order for this partition to be stored in an array with no explicit node pointers we must choose the median position such that the tree will be left balanced (**?**), Algorithm 2 describes the procedure necessary for this to be the case.

Algorithm 2 Balanced Median Calculation

$n \leftarrow 0$
while $2^{n+1} <= N$ **do**
    n $\leftarrow$ n + 1
end while
$M \leftarrow 2^n$
$R \leftarrow N - (M - 1)$
if $R <= M/2$ **then**
    return $(M - 2)/2 + R$
else
    return $(M - 2)/2 + M/2$
end if

5.6 Raytracing

We will now discuss the implementation of the ray-tracer used in the system to find the intersection points and shading specular surfaces. Multiple ray-tracing threads are created, these threads are passed a pointer to a queue that when read returns a pixel coordinate that the thread will trace, this queue is written by an additional thread that will write each pixel in the output image as space becomes available in the queue, the queue is of sufficient size so as the ray-tracing threads do not exhaust the thread. In order to signal to the ray-tracing thread that the render has finished a flag can be set with the pixel data that indicates that the thread should finish executing. One finish signal is sent to each of the threads.

Algorithm 3 Raytracing Main Thread

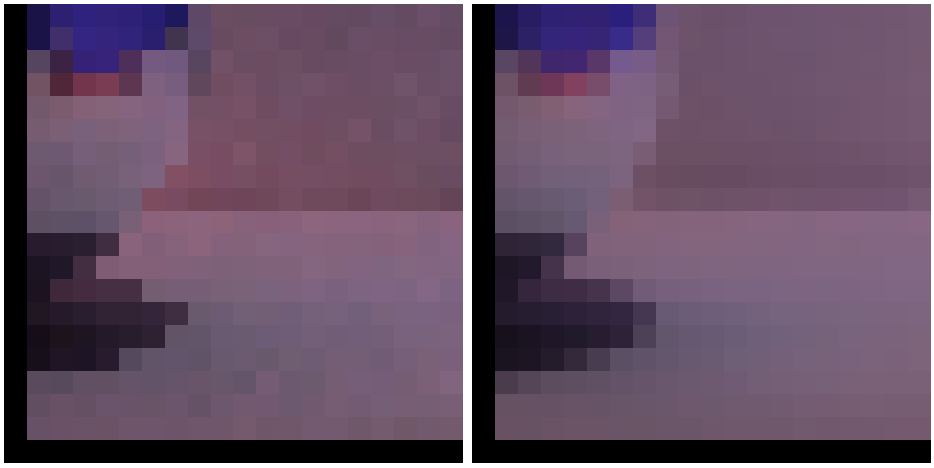
```
for 0 to num_threads do
    START_THREAD(input_queue, output_queue)
end for
for 0 to num_pixels do
    pixel_data ← QUEUE_READ(output_queue)
    UPDATE_PIXEL(pixel_data)
end for
```

Algorithm 4 Raytracing Processing Thread

```
pixel_data ← QUEUE_READ(input_queue)
while pixel_data.continue do
    colour ← black
    for each sample in pixel do
        colour += TRACE_PIXEL
    end for
    QUEUE_WRITE(output_queue, pixel_data.x, pixel_data.y, colour)
    pixel_data ← QUEUE_READ(input_queue)
end while
```

5.6.1 Multisampling

For each pixel that a ray-trace thread receives multiple rays are traced through the scene in order to perform multisampling on the pixel, this allows us to remove aliasing in the final image and perform distribution ray-tracing, we perform stratified sampling of the pixel area to calculate the location of each of the sub-pixels.



(a) 1 sample per pixel (b) 25 samples per pixel

5.6.2 Scene Traversal

Each ray that is traced in the scene requires that we traverse the scene in order to find the closest intersection, in this system we store the objects as a list of pointers to there respective data, as a result of this the scene traversal algorithm is quite simple, we iterate over each of the objects and test if the ray intersects, if not then we continue, if so we check the intersection t parameter with that of the closest intersection found up to that point (initially set to ∞). If an intersection is found we then call the shade function of the intersecting objects in order to find the radiance value for this ray, otherwise we set the colour to a default sky colour found in the scene description.

Algorithm 5 Scene Traversal Algorithm

```
colour ← scene.sky_colour
record.t ← ∞
for object ∈ scene do
    new_record ← INTERSECTS(ray, object)
    if new_record.hit and new_record.t < record.t then
        record ← new_record
    end if
end for
if record.hit then
    colour ← SHADE(scene, record)
end if
return colour
```

Self Intersection

Due to the floating point errors in the calculation of intersection tests the point of intersection that is reported may be slightly incorrect, this inaccuracy can cause issues for reflection, transmission and shadows as the intersection of the specular or shadow ray reports an intersection point close to the original intersection point, this can cause artefacts in the final render, to remedy this issue we use a small offset in the direction of the incident ray so that the origin of the new ray is slightly above the intersection point.

Volume Intersection

Unlike intersection with all other object types the intersection with a participating media is non deterministic as the media represents a stochastic distribution of particles as a result an intersection test with the same ray may return a different intersection point if performed multiple times, this property allows us to perform multiple samples that will converge to the correct appearance of the participating media allowing affects such as objects embedded within the media to be visible but occluded by the media.

Intersection Records

Each intersection that is performed in the system gathers certain data that is needed by later stages of the system, for example an intersection with a triangle will not only return the t parameter for the intersecting ray but also barycentric coordinates that can be used to interpolate properties of the triangle vertices across the surface of the triangle. In order to store this information a pointer to a intersection records is passed as a parameter for each intersection test, this intersection record is then used during the shading section of the ray-tracing algorithm.

5.7 Shading

Once we have found a point of intersection we must now calculate the colour at that point, this will take into account the reflectance properties of the object and the photon map surrounding it, this will in effect be an evaluation of the rendering equation, which we will evaluate by splitting the equation into several components, these components are direct illumination, specular reflection and transmission which we evaluate using ray-tracing and diffuse inter-reflection and caustics which will use the photon map to estimate the radiance.

Recall from Chapter 2 the rendering equation is given in terms of incoming and outgoing radiance, omitting emitted radiance we can rewrite the rendering as the sum of seperate integrals representing different light paths in the scene. (**?**)

$$\begin{aligned} L_r(x,\omega) = &\int_\Omega f_r(x,\omega,\omega')L_{i,l}(x,\omega,\omega')(\omega\cdot n)d\omega' + \\ &\int_\Omega f_{r,S}(x,\omega,\omega')(L_{i,d}(x,\omega,\omega') + L_{i,c}(x,\omega,\omega'))(\omega\cdot n)d\omega' + \\ &\int_\Omega f_{r,D}(x,\omega,\omega')L_{i,c}(x,\omega,\omega')(\omega\cdot n)d\omega' + \\ &\int_\Omega f_{r,D}(x,\omega,\omega')L_{i,d}(x,\omega,\omega')(\omega\cdot n)d\omega' \end{aligned}$$

where $L_{i,c}$ is the irradiance due to **LS+D** paths, $L_{i,d}$ due to **L(S|D)$^+$D** paths and $L_{i,l}$ irradiance directly from light sources (**LD** paths).

5.7.1 Radiance Estimations

Before we discuss how we utilise the photon map to approximate the global illumination of a scene we must first discuss how we use the photon map to estimate the radiance at a point. Recall from Chapter 2 the radiance at a point of intersection can be estimated by,

$$L(x,\omega) \approx \sum_{n=1}^{N} f_r(x,\omega,\omega'_n)\frac{\Delta\Phi_n}{\pi r^2} \tag{5.3}$$

where $f_r(x,\omega,\omega')$ is the BRDF for the surface, we will only use the photon map at diffuse surfaces, as the BRDF for these surfaces are a constant we can move this calculation out of the summation to give us the following.

$$L(x,\omega) \approx \frac{\rho_d}{\pi}\sum_{n=1}^{N}\frac{\Delta\Phi_n}{\pi r^2} \tag{5.4}$$

It can be seen that we are now estimating the irradiance at the point of intersection, to use the photon map then, we need to find the N nearest photons to the point at which we are estimating the radiance, to do this we perform the nearest neighbour algorithm for k-d trees on the photon map. The algorithm begins at the root of the tree, we calculate the distance of the point to the split plane at this point in the tree, if the point is less than the split plane we recursively call the algorithm with the left tree else the right, if after the recursive call has returned the distance to the splitting plane is less than the search radius we recursively call the algorithm on the other child node this continues until we are at a leaf in the tree, at this point we calculate if the photon is within the search radius, if so we insert the photon into a list of photons, if we have already found the number of photons the photon that is furthest from the point is discarded.

Algorithm 6 K-D tree N Nearest Neighbours algorithm

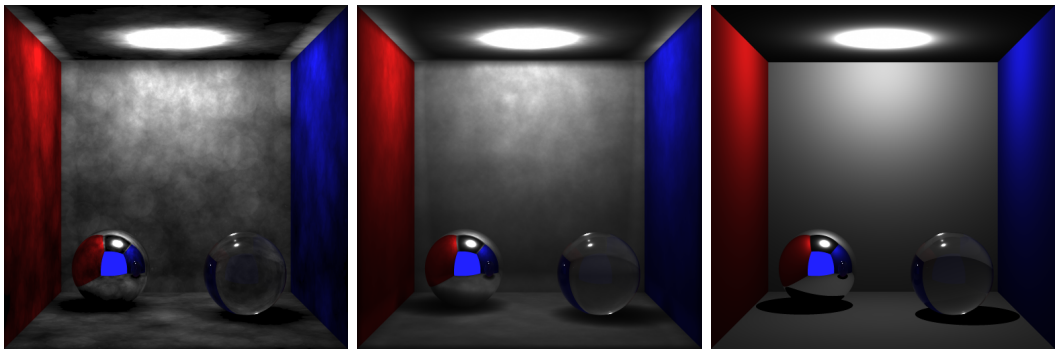
```
function LOCATE(p)
    n ← number of photons
    δ ← signed distance to splitting plane
    r ← distance to furthest photon found
    if 2p + 2 < n then
        if δ < 0 then
            LOCATE(2p + 1)
            if δ² < r² then
                LOCATE(2p + 2)
            end if
        else
            LOCATE(2p + 2)
            if δ² < r² then
                LOCATE(2p + 1)
            end if
        end if
        if δ² < r² then
            Insert photon into list
        end if
    end if
end function
```

5.7.2 Direct Illumination

$$\int_{\Omega} f_r(x, \omega, \omega') L(x, \omega, \omega')_{i,l} (\omega \cdot n) d\omega'$$

Direct illumination is the radiance contribution at a surface directly from the light sources in the scene (**LDE** in path notation) this term is responsible for fine details such as shadows in the scene, as a result we use ray-tracing to calculate the radiance due to direct illumination,

it is possible to calculate the radiance due to this term from the photon map but this approach leads to noise in the final image even when using a high number of photons in the radiance estimate, a comparison of direct illumination with ray-tracing and the photon map can be seen in Figure 5.10 it can be seen that the illumination calculated by both methods are largely similar but there is significant variance in the appearance in the photon mapped image, we can also see that areas close to corners have a lighter appearance, this is due to the use of a sphere to gather photons in the nearest neighbour search as a result photons that do not belong to a surface are used as part of the estimate.



(a) 25,000 photons (b) 500,000 photons (c) Raytraced

Figure 5.10: Comparison of the photon map radiance estimate with direct illumination

The system currently supports two types of light, point lights and area light, we will first consider calculating the radiance from a point light as it is the simpler of the two cases.

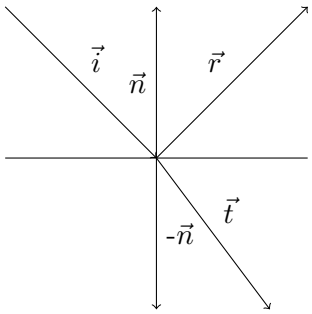
The radiance from a point light is constant for all points at the same distance from the origin of the light, the radiance at a point in the scene is given by Equation (5.5), where the function V is the visibility function, that is if the point of intersection cannot see the point light it will not contribute the radiance at that point, this will cause the appearance of shadows in the render.

$$E(x) = V(x, s)\frac{\Phi_s \cos\theta}{4\pi r^2} \tag{5.5}$$

In the case of area lights calculating the radiance at a point is more complicated, we must calculate the fraction of the area of the light that is visible at the point of intersection, to do this we sample a number of shadow rays that evaluate the visibility function across the area of the light, to do this we approximate the area light as $n \times n$ point lights.

Texture Mapping

Performing texture mapping requires the ability to query the texture coordinates at a point of intersection, for mesh objects this will interpolate the barycentric coordinates for the



triangle of intersection, spheres use a spherical mapping that uses the spherical coordinates to generate u,v values, we use linear sampling to calculate the colour from the texture.

5.7.3 Specular Reflection and Transmission

$$\int_{\Omega} f_{r,S}(x,\omega,\omega')(L_{i,d}(x,\omega,\omega') + L_{i,c}(x,\omega,\omega'))(\omega \cdot n)d\omega'$$

For specular surfaces we again use ray-tracing to evaluate the contribution, as mentioned previously the BRDF for specular surfaces contains two delta functions (one for incoming ray and one for outgoing ray) as a result we cannot use the photon map to evaluate the contribution from specular surfaces, we will again use ray-tracing to evaluate this component by tracing an additional reflected or refracted ray.The equations used to calculate these rays are given in Equations (5.7) and (5.8) where η_1 and η_2 are the refractive index of the medium we are leaving and entering respectively.

$$\cos\theta_i = -\vec{i} \cdot \vec{n} \qquad \text{and} \qquad \sin^2\theta_t = \left(\frac{\eta_1}{\eta_2}\right)^2 (1 - \cos^2\theta_i) \tag{5.6}$$

$$\vec{r} = \vec{i} + 2\cos\theta_i \vec{n} \tag{5.7}$$

$$\vec{t} = \frac{\eta_1}{\eta_2}\vec{i} + \left(\frac{\eta_1}{\eta_2}\cos\theta_i - \sqrt{1 - \sin^2\theta_t}\right)\vec{n} \tag{5.8}$$

Fresnel Reflection

Transmissive surfaces reflect a proportion of the radiance incident at the surface as it passes from a material with a different index of refraction, this proportion is determined by the Fresnel equation for dielectrics, which is a function of the refractive index of the two materials and the incident angle, this is given by equation 5.9

$$R_f(\theta) = \frac{\left(\frac{\eta_2 \cos\theta_i - \eta_1 \cos\theta_t}{\eta_2 \cos\theta_i + \eta_1 \cos\theta_t} + \frac{\eta_1 \cos\theta_i - \eta_2 \cos\theta_t}{\eta_1 \cos\theta_i + \eta_2 \cos\theta_t}\right)^2}{2} \tag{5.9}$$

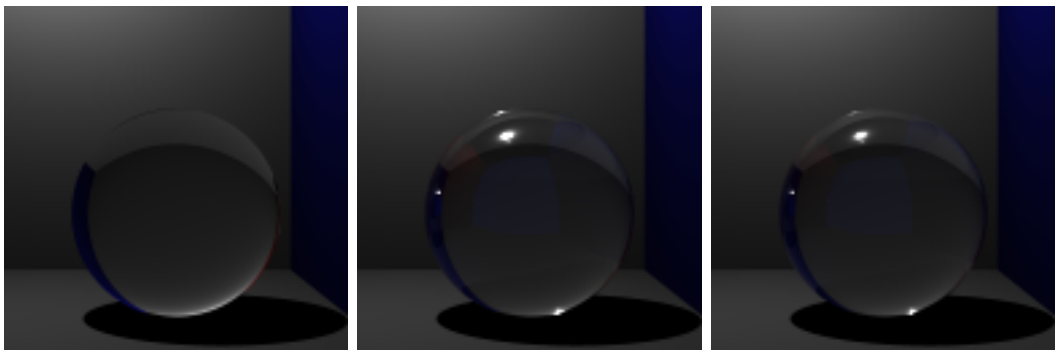
Schlick Approximation

Calculating the Fresnel term for each refracted ray calculation can be costly, we can however use an approximation of the Fresnel term by Schlick (**?**) this is given by:

$$R_s(\theta) = R_0 + (1 + R_0)\,(1 - \cos\theta)^5 \tag{5.10}$$

where R_0 is the Fresnel reflectance value for a ray parallel to the surface normal which can be precomputed, this has been been shown to be up to 30% faster (**?**) than calculating the Fresnel term for each refracted ray, Figure 5.11 demonstrates the affect of accounting for the Fresnel term and also using the Schlick approximation.

As with other aspects of the system where multiple rays can be spawned from a surface interaction (in this case one reflected and one refracted ray) we use Russian roulette in order to determine if we reflect or refract.



(a) no Fresnel reflection (b) Fresnel reflection (c) Schlick reflection

Figure 5.11: Demonstration of Fresnell reflection and the Schlick approximation

5.7.4 Diffuse Interreflection

$$\int_\Omega f_{r,D}(x, \omega, \omega') L_{i,d}(x, \omega, \omega')(\omega \cdot n) d\omega'$$

Diffuse interrefection is the contribution that occurs from photons that have been bounced from a diffuse surface at least once ($\mathbf{LD}(\mathbf{S}|\mathbf{D})^+\mathbf{E}$) while it is possible to estimate the contribution to the radiance at an intersection point directly from the photon map this approach can cause visual artefacts due to variance in the estimate. In order to reduce

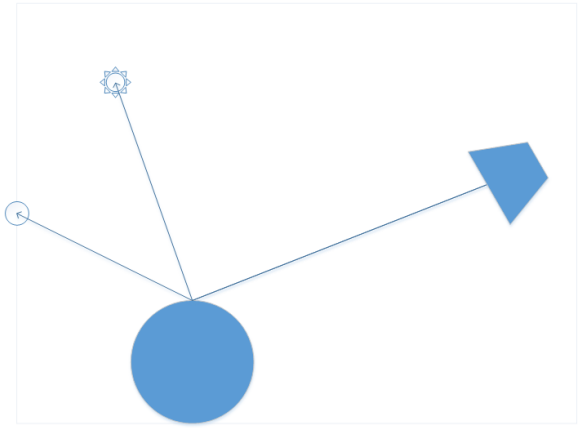


Figure 5.12: Final Gather

these artefacts we perform a final gather stage at the point of intersect that produces multiple diffuse ray that is traced into the scene until a non-specular object is intersected, we then perform the radiance estimate at this point and use this information to estimate the radiance incident at the original point of intersection. In order for the final gather to produce a correct estimate of the radiance we need to perform this stage multiple times per pixel. As we are performing distributed ray-tracing this is a trivial addition.

5.7.5 Caustics

$$\int_{\Omega} f_{r,D}(x,\omega,\omega')L_{i,c}(x,\omega,\omega')(\omega \cdot n)d\omega'$$

Caustics occur due to the focusing effect of curved specular surfaces, in other global illumination algorithms caustics have been hard to simulate (**?**) and often cause large amounts of noise in the final image with the photon map we are able to estimate the radiance directly from the caustic photon map, as we have separated the caustic photons from all other photon paths we reduce the nose that is introduced by caustics in all other radiance estimates using the photon map. As caustics generally cause sharp visual effects using to few photons in the radiance estimate can cause unwanted blurring (**?**), in order to reduce this we use a filter as part of the radiance estimate such that photons near the query point contribute more to the radiance estimate, Jensen suggests the use of one of two filters, the cone filter and the Gaussian filter, each requiring a modification to the photon mapping radiance estimate.

Cone Filter

We use the cone filter by applying a weight w_{pc} based upon the distance of the photon from the query point, this will cause photons that are closer to the point to contribute more to

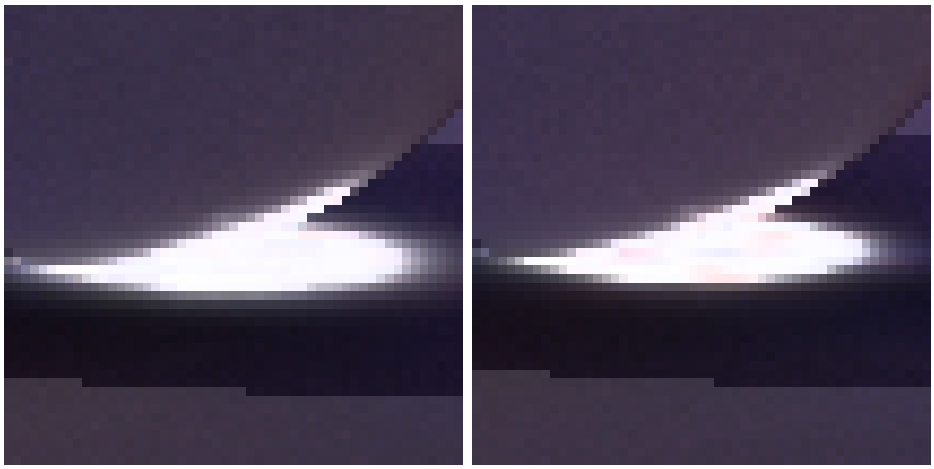
the radiance estimation.

$$w_{pc} = 1 - \frac{d_p}{kr} \tag{5.11}$$

where k is the filter parameter, d_p the distance of the photon from the query point and r the maximum radius of all photons found in the nearest neighbour search, the updated radiance estimate for the cone filter is given by:

$$L_r(x, \vec{\omega}) \approx \frac{\sum_{p=1}^{N} f_r(x, \vec{\omega}_p, \vec{\omega}) \Delta\Phi_p(x, \vec{\omega}_p) w_{pc}}{(1 - \frac{2}{3k})\pi r^2} \tag{5.12}$$

the term $(1 - \frac{2}{3k})$ is a normalisation factor for the cone filter (**?**)



(a) without the cone filter (b) with the cone filter

Figure 5.13: Comparison with and without cone filter

5.8 Participating Media

In the case of a ray that intersects a participating media before intersecting a surface we need to calculate three components to the radiance along the path of the ray in the medium, these are single scattering direct illumination, multiple scattering (in-scattering) and attenuation (out-scattering)

5.8.1 Ray Marching

In order to evaluate the radiance from the participating media we perform a ray-march, this is an iterative operation that evaluates the radiance along the ray as it moves through the participating media.

$$L(x,\omega) = \sum_{i=1}^{N} L_l(x,\omega_l')p(x,\omega_l',\omega)\sigma_s(x)\Delta x + e^{-\sigma_t \Delta x} L(x+\omega\Delta x,\omega) \tag{5.13}$$

5.8.2 Attenuation

As a ray travels through a medium the radiance can be reduced due to out-scattering and absorption, this can be calculated by evaluating the integral given in Equation (5.15), as we are only considering homogeneous participating media this can be simplified as the properties being integrated are constant and a closed form solution is possible, the attenuation for any participating media is given by:

$$e^{-\tau(x,x+\Delta x)} \tag{5.14}$$

where τ is given by:

$$\tau(x,x') = \int_{x}^{x'} \sigma_t(t)dt \tag{5.15}$$

as $\sigma(t)$ is a constant for participating media we can evaluate this integral and use the result in Equation (5.14) to give the attenuation of the radiance along the distance Δx as:

$$\tau(x,x+\Delta x) = \int_{x}^{x+\Delta x} \sigma_t.dt = \sigma_t \Delta x$$

5.8.3 Direct Illumination

At each point in the ray march we also evaluate the contribution from each light in the scene due to single scattering, this is performed by performing an additional ray march in the direction of the light and evaluate the radiance arriving at the point on the ray.

5.8.4 Multiple Scattering

In order to evaluate the effect of multiple scattering within the participating media we use the photon map and the volume radiance estimate for the photon map, as with direct illumination we evaluate the in-scattering at discrete points along the path of the ray.

Chapter 6

Testing

Testing of the program was performed in two main ways, testing of the individual functions and testing of the system as a whole. Certain functions designed to accelerate the ray-tracing need to be compared to a base implementation that while slower is guaranteed to be correct, for instance a finding the closest intersection of a ray and mesh, the simplest way to do this is to test the ray and every triangle in the mesh and record the closest intersection, this is simple to perform and the scope for errors in the implementation is low, unfortunately this is a very expensive operation that requires $O(n)$ time complexity. In this project the k-d tree is used to accelerate the test, this can be performed in $O(log(n))$ time complexity, the code to perform the intersection test for this data structure is much more complicated and as a result errors in the implementation are more likely. Testing of these complicated functions was done by comparing the output of the simple function with that of the faster implementation for identical data, it can also be used to see if the new implementation does indeed increase the performance of the code. Figure **??**. For other aspects of the system full testing was not possible due to time constraints, in these cases we used a heuristic approach of testing whereby we ran the system with test scenes and compared the results of to out expectation of the final image.

6.1 System Testing

6.1.1 Photon Viewer

As the development of the system was performed roughly in the order described in the document, the photon map was produced before the system used it in the radiance estimate when creating the images, as a concequence verifiying the results of the photon map generator posed a particular challenge as it typically contained thousands of photons in a geometry independent data structure, as part of the development of the product we created a companion utility to the main system that allows for the visualisation of the photon map, when inputted into the program we can see if the results match the expectation of

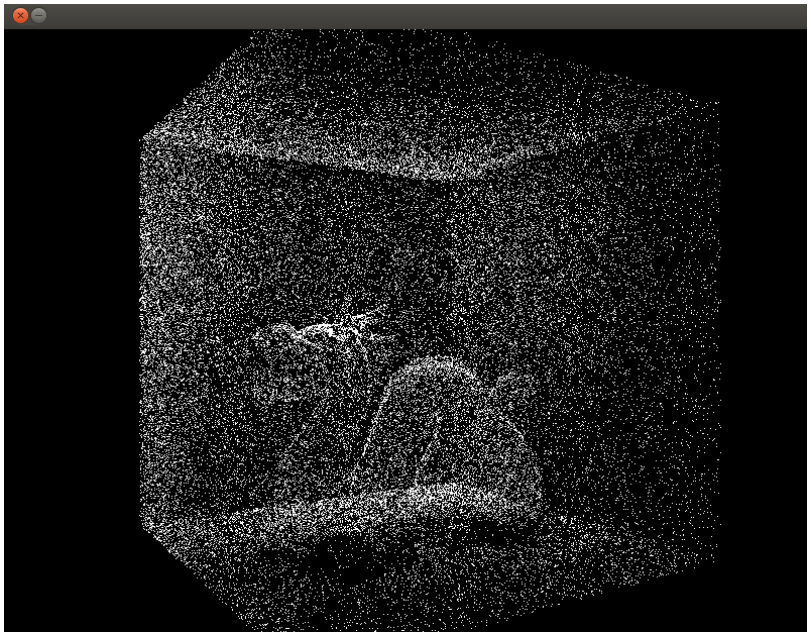


Figure 6.1: Photon Viewer Running

the distribution of the photons for a given scene. The photon viewer is reads in data from a simple format containing as a first line the number of photons and then a list of the same length of photon positions, powers and incident directions. This tool proved to be of great value also being used to test the sampling functions. The design of the photon viewer is similar to that of the main GUI, differing in that it displays points stored in an OpenGL display list (**?**) (used to lower the number of draw calls) as oppose to rendering a texture of the output image as in the main GUI.

6.1.2 Pixel Tracing

When performing tests on the system it would frequently be the case that a change introduced obvious error in the output image, it became apparent early in the development that being able to trace a single pixels path through the scene would make the task of debugging that much easier. This functionality is available as the `--trace_pixel` option.

6.2 Performance Testing

During the design and implementation of the system we have stated that certain decisions were made in order to improve the performance of the system, in this section we will present results of testing that was performed in order to evaluate the gains from these decisions in a concrete way.

6.2.1 K-D tree intersection acceleration

We stated in Chapter 5 that we used triangle k-d trees to accelerate the intersection test for meshes, this test is designed to test the gains that we saw when using the k-d tree, the test was performed by running the system on a scene with a complex model (the Stanford dragon) with and without the k-d tree intersection test, for the case without the k-d tree the intersection test is performed by testing each triangle in the mesh and recording the closest intersection. For this test we have disabled photon mapping so as to only test the ray-tracing part of the system, this was done through prepossessing defines in the system at compile time.

6.2.2 Multicore Scaling

In Chapter 3 and 4 we highlighted the inherent parallelism possible in the photon mapping algorithm in both the photon generation and ray-tracing stages, in order to test the performance of the system and its ability to scale with more resources we decided to acquire access to Amazon EC2 compute clusters, allowing us to run the system on up to 32 Intel Xeon E5-2670 cores and 60GB of RAM (**?**), we tested the system with various settings such as resolution and number of photons in the photon map and in the radiance estimate , each configuration will be run multiple times in order to reduce variance due to the hardware. We shall test two parts of the system in isolation to gauge how well they have been parralised the first is the photon map generation and the second the ray-tracing stage, in order to test them in isolation we modify the system such that in the first case it exits after creating the photon map and in the second the photon maps are not created at all.

Chapter 7

Results

In this Chapter we will present the results of the testing that we performed on the system and present output images from the system that will demonstrate and highlight various effects that we are able to simulate with the system.

7.1 K-D Tree Performance

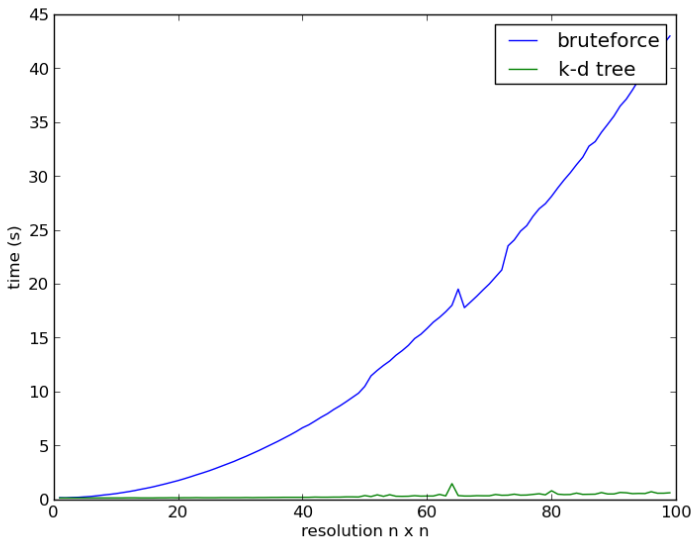


Figure 7.1: Comparison of bruteforce and k-d tree mesh intersection test

It can be seen in Figure 7.1 that there are clear gains had from using the k-d tree when performing the intersection test, this test was performed for image resolutions from 1×1

to 100×100, even for these low resolution tests we can see how infeasable performing the bruteforce method becomes. It is clear that implementing the k-d tree acceleration structure is of great benifit to the system.

7.2 Multicore Scaling

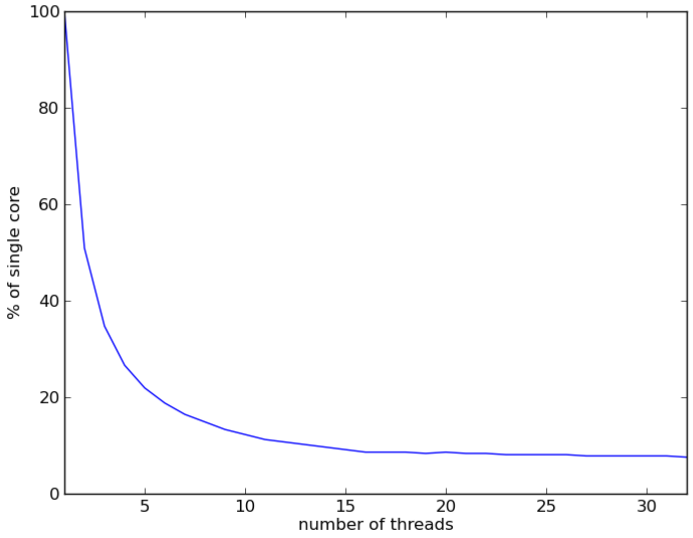


Figure 7.2: Raytracing Parrelism Test

It can be seen from Figure 7.2 that performance gains were acheived as cores were added to the system, these results do show that this part of the system is able to utilise the hardware of even relitivly powerful machines, we can see that we begin to see diminishing returns on the speedup, while it would be desirable to improve the system to further utilise the resources of the machine for the most common use case of a machine with a low number of cores we see a near linear speedup.

The results for the photon map construction test were less impressive, it can be seen in Figure 7.3 that for large numbers of cores the performance of this part of the system begins to deterierate, we were not able to in the time frame of the project able to determin the cause of this behavoir, not least because for core counts of less than five we see that the system does indeed improve and as the main development machine only has four cores debugging on this machine was not an option. We beleive that the design of this componenet of the system was perhaos naive in its approach to parralism and a more sophisticated desing was needed.

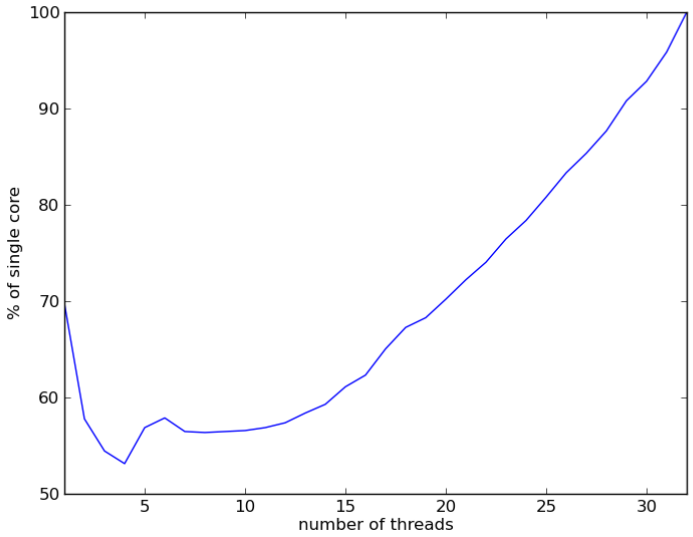
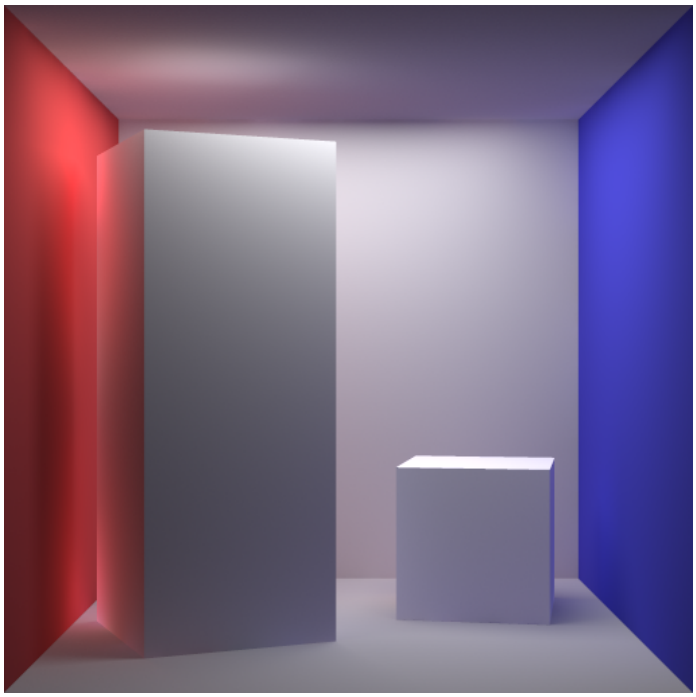


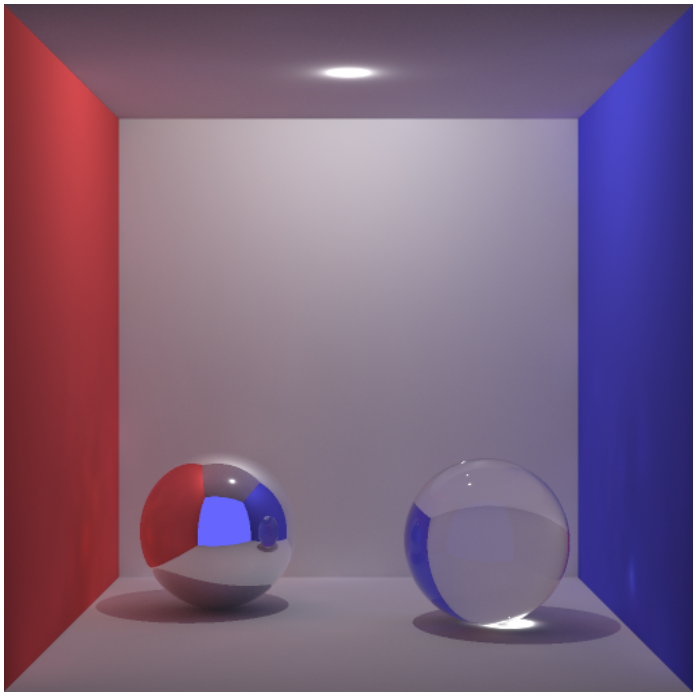
Figure 7.3: Photon map construction parrelism test

7.3 System Output

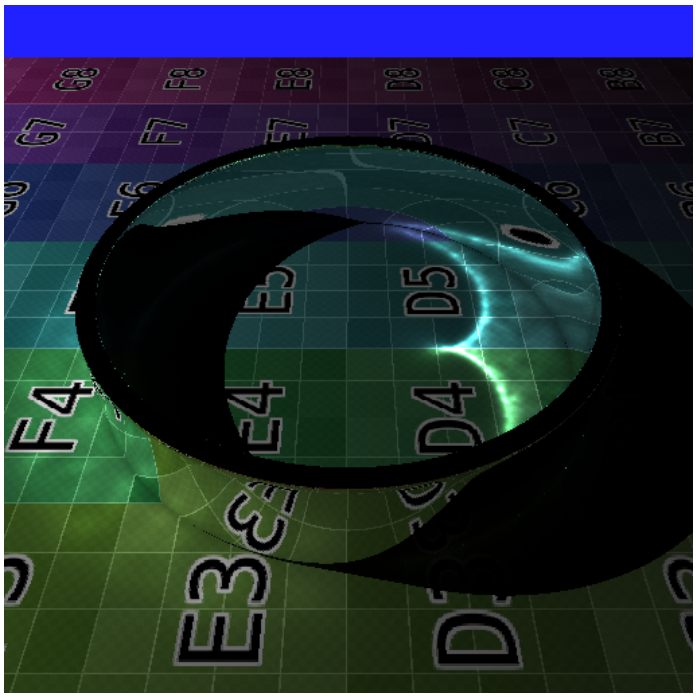
In this section we shall present images that were created with the system to showcase the final product, each rendered scene displays the use of the photon mapping algorithm and how we have been able to use it to create visually pleasing images.



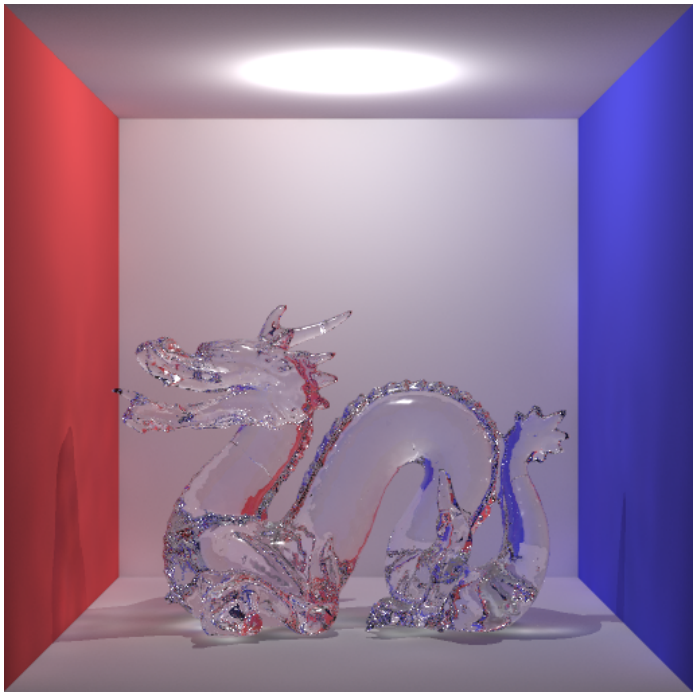
(a) Cornell Box (note diffuse reflection of the walls and top of the box on the ceiling)



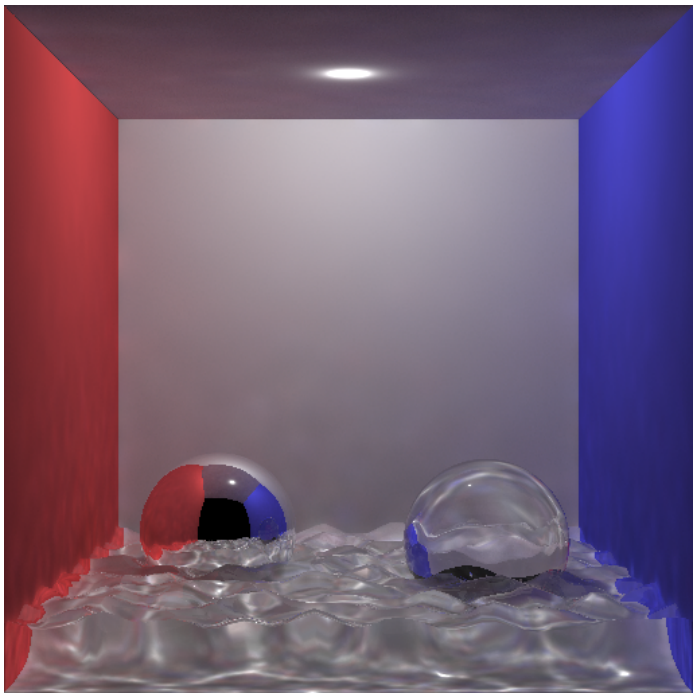
(b) Cornell Spheres (note caustics on the glass sphere)



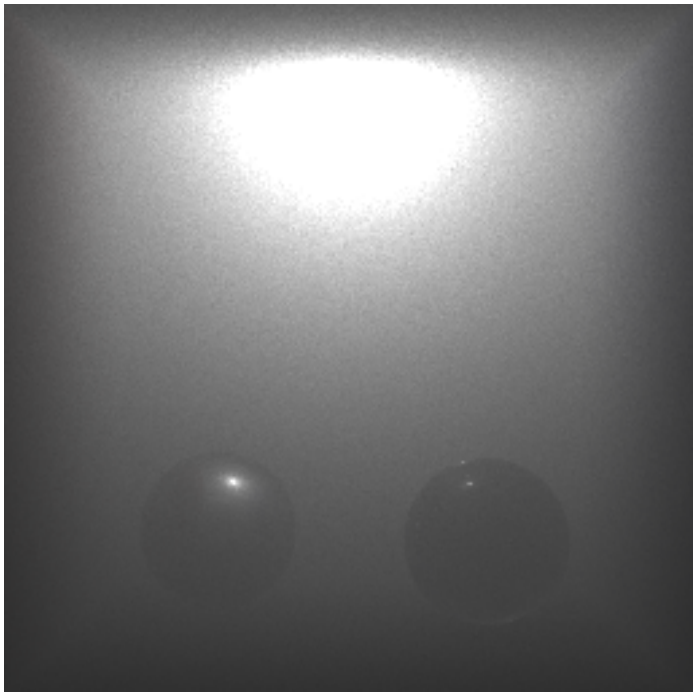
(c) Caustic Ring



(d) Stanford Dragon



(e) Cornell box filled with water (note caustic ripples on the floor)



(f) Cornell box filled with smoke

Chapter 8

Conclusions

We described in the introduction that the aim of this project was to produce a system that performed the photon mapping algorithm in order to create interesting images that simulate real-world phenomena, we described in the literature review the work that inspired and informed this projects as well as techniques that have aided us during the development of the system, these works allowed us to create a design that we feel we were able to implement faithfully and has resulted in a system that fulfil the aims of this project. We shall now briefly discuss the system and give some final thoughts.

8.1 Requirement Assessment

We feel that we have been able to fully satisfy most of the requirements that we set forth in Chapter 3, all functional requirements where met while most of the non-functional requirements were met, those that were not we feel that the exclusion of the requirement from the final product is not detrimental to the aims of the project, in particular we state in requirement 2.6 that the output of the system should be consistent across runs, this proved to be difficult to achieve as we are frequently dealing with stochastic phenomena and utilising multiple threads, as a result we were not able to satisfy this requirements. We also were unable to test the system on a Mac OSX system as we were unable to acquire suitable hardware although we feel due to the similarities in design of Linux and Max OSX (both UNIX derivatives) they system should work unaltered.

8.2 Deviation from Original Plan

Through the course of the project the scope it was necessary to evaluate the scope and direction of the project, the initial plan of the project was to include an alteration to the participating media photon mapping algorithm to increase the efficiency of the radiance estimate for certain light types (simulation of laser light) through the development of the

project the complexity of implementing a system with participating media and details of the system not considered during the initial planning stages and investigating the suitability of the new approach in light of these details we decided to omit the algorithm from the final product of this project.

8.3 Evaluation and Future Work

The system that has been produced delivers upon the description of this project, we have shown that we are able to produce images of a wide range of effects that are not possible with traditional ray-tracing and are able to do so in a time that is significantly lower than that of a solution such as path tracing. While the aims of the projects were met there are a few aspects of the project that we feel could have been improved, for instance we recognise that during the scene traversal performing the intersection test on each object is inefficient and that acceleration structures such as bounding volume hierarchies would have served to increase the performance of the system, it was however decided that given the time constraints of the project that development effort was better spent on accelerating aspects such as the intersection test for triangle meshes as this would provide us with a larger performance gain for the majority of scenes that we tested the system with which frequently contained a small number of objects with a high triangle count. From the testing we could see that performance of the threaded photon map generation was poor, we were not able to pinpoint the cause of this degradation of performance for thread counts of more than five.

As with any project time constraints determine the number of features that can be reasonably be added, this includes algorithms that would make the system more efficient such as irradiance caching. We discussed in Chapter 2 advances in the photon mapping algorithm since its inception, we feel that it would make an interesting investigation to extend that current system to include some of these techniques. If we were to do this project again we feel that we would have conducted the investigation and development differently, specifically we would have performed the performance continuously throughout the development so at to catch issues such as the photon map generation early and perhaps could have rectified the issue. Given more time with the project we also would have liked to include a more general description of the BRDF so as more interesting materials such as brushed metals etc could be simulated.

8.4 Final Thoughts

As a conclusion to my degree studies this project has taught me a great deal and serves as a document of the progression in my technical and non-technical skills, additionally the research performed in preparation of the project and for the duration has showed my the how areas of seemingly unrelated fields can come together to create some of the results described in this project, such as monte carlo methods first used during the Second World

War as part of the allied efforts to construct the first nuclear weapon. Overall we feel that this project has been a success and that we have managed to do what we set out to do and are pleased with the results that we were able to achieve.

Bibliography

Amazon (2014), 'Amazon ec2 instance types', "`http://aws.amazon.com/ec2/instance-types/`".

Appel, A. (1968), Some techniques for shading machine renderings of solids, *in* 'AFIPS 1968 Spring Joint Computer Conference', Vol. 32, pp. 37–45. first ray tracing paper, light ray tracing, b&w pictures on Calcomp plotter.

Baerentzen, A. (2003), '"on left-balancing binary trees', `http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/2535/pdf/imm2535.pdf`.

Bentley, J. L. (1975), 'Multidimensional binary search trees used for associative searching', *Communications of the ACM* **18**(9), 509–517.

Bjorkman, W. W. and Wolff (2001), 'Introduction to monte carlo radiative transfer', `http://www-star.st-and.ac.uk/~kw25/research/montecarlo/book.pdf`.

Blinn, J. F. (1982), 'Light reflection functions for simulation of clouds and dusty surfaces', *SIGGRAPH Comput. Graph.* **16**(3), 21–29.
URL: http://doi.acm.org/10.1145/965145.801255

de Greve, B. (2006), 'Reflections and refraction in ray tracing', `http://graphics.stanford.edu/courses/cs148-10-summer/docs/2006--degreve--reflection_refraction.pdf`.

Foley, J. D. (1997), *Computer graphics: principles and practice*, Addison-Wesley Systems Programming Series.

Friedman, J. H., Bentley, J. L. and Finkel, R. A. (1977), 'An algorithm for finding best matches in logarithmic expected time', *ACM Transactions on Mathematical Software* **3**(3), 209–226.

Fussell, D. and Subramanian, K. R. (1988), Fast ray tracing using K-D trees, Technical Report TR-88-07, U. of Texas, Austin, Dept. Of Computer Science.

Goral, C. M. (1985), A model for the interaction of light between diffuse surfaces, Master's thesis, Cornell University.

Hachisuka, T. and Jensen, H. W. (2009), Stochastic progressive photon mapping, *in* 'ACM SIGGRAPH Asia 2009 Papers', SIGGRAPH Asia '09, ACM, New York, NY, USA, pp. 141:1–141:8.
URL: http://doi.acm.org/10.1145/1661412.1618487

Hachisuka, T., Ogaki, S. and Jensen, H. W. (2008), 'Progressive photon mapping', *ACM Transactions on Graphics* **27**(5), 130:1–130:??

Jensen, H. W. (1996), Global illumination using photon maps, *in* '7th Eurographics Workshop on Rendering', pp. 22–31.

Jensen, H. W. (2001), *Realistic Image Synthesis Using Photon Mapping*, A. K. Peters, Natick, MA.

Jensen, H. W. and Christensen, N. J. (1995*a*), Efficiently rendering shadows using the photon map, *in* 'Proceedings of Compugraphics '95', pp. 285–291.

Jensen, H. W. and Christensen, N. J. (1995*b*), 'Photon maps in bidirectional monte carlo ray tracing of complex objects', *Computers & Graphics* **19**(2), 215–224.

Jensen, H. W. and Christensen, P. H. (1998), Efficient simulation of light transport in scenes with participating media using photon maps, *in* 'Computer Graphics (ACM SIGGRAPH '98 Proceedings)', pp. 311–320. r.

Jensen, H. W., Marschner, S. R., Levoy, M. and Hanrahan, P. (2001), A practical model for subsurface light transport, *in* E. Fiume, ed., 'SIGGRAPH 2001, Computer Graphics Proceedings', Annual Conference Series, ACM Press / ACM SIGGRAPH, pp. 511–518.
URL: http://visinfo.zib.de/EVlib/Show?EVL-2001-143

Kajiya, J. T. (1986), The rendering equation, *in* 'Proceedings of Siggraph '86', pp. 143–150.

Khronos-Group (2014), 'OpenGL FAQ : Should I use display lists', `https://www.opengl.org/wiki/FAQ#Should_I_use_display_lists.2C_vertex_arrays_or_vertex_buffer_objects.3F`. [Online; accessed 10-April-2014].

Knuth (1974), 'Computer programming as an art', *CACM: Communications of the ACM* **17**.

MacDonald, J. D. and Booth, K. S. (1990), 'Heuristics for ray tracing using space subdivision', *The Visual Computer* **6**(3), 153–66.

Nicodemus, F. E. (1965), 'Directional reflectance and emissivity of an opaque surface', **4**, 767.

QNX (2014), '"programming tools - opaque pointers"', `http://www.qnx.com/developers/articles/article_302_2.html`.

Schlick, C. (1994), 'An Inexpensive BRDF Model for Physically-Based Rendering', *Computer Graphics Forum* **13**(3), 233–246.

Shirley, P. (2003), 'Distribution raytracing: Theory and'.
URL: http://citeseer.ist.psu.edu/623047.html; http://www.cs.utah.edu/ shirley/papers/rw92.pdf

Wald, I. and Havran, V. (2006), On building fast kd-trees for ray tracing, and on doing that in o(n log n), *in* 'IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING', pp. 61–70.

Ward, G. J., Rubinstein, F. M. and Clear, R. D. (1988), 'A ray tracing solution for diffuse interreflection', *SIGGRAPH Comput. Graph.* **22**(4), 85–92.
URL: http://doi.acm.org/10.1145/378456.378490

Weiss, M. and Grosch, T. (2012), 'Stochastic progressive photon mapping for dynamic scenes', *Comp. Graph. Forum* **31**(2pt4), 719–726.
URL: http://dx.doi.org/10.1111/j.1467-8659.2012.03051.x

Whitted, T. (1979), An improved illumination model for shaded display, *in* 'An improved illumination model for shaded display', Vol. 13, pp. 1–14.