# Decoupling Illumination From Isosurface Generation

Steven Challis

Computing BSc

2009/2010

The candidate confirms that the work submitted is their own and the appropriate credit has been given where reference has been made to the work of others.

I understand that failure to attribute material which is obtained from another source may be considered as plagiarism.

(Signature of student)___________________________

# Summary

This project is a study of current methods to precompute global illumination for the purpose of interactively lighting isosurfaces. A recent paper has improved upon previous methods by decoupling the illumination phase from isosurface generation thereby negating the need to repeatedly extract and light isosurfaces. This is important because it enables the infinite set of all possible isosurfaces contained within a volume to be lit simultaneously using uniform sampling of the radiance values. A working implementation of this new method will be presented and the novel reformulation of light transport underpinning it will be discussed in the wider context of computer generated illumination. The design decisions and problems encountered are also covered. Further, an attempt has been made to explore sampling and interpolation techniques.

# Acknowledgements

My thanks go first and formost to my project superviser, Dr. Hamish Carr for his invaluable assistance throughout this project, and for the thought provoking discussions he was always happy to have despite his busy schedule. I would also like to thank my assessor Dr. Roy Ruddle for the critical feedback he provided on both my mid-project report and during the progress meeting. The support of my friends and family does not go unnoticed and I am endlessly grateful for everything they have done for me, not just during this project, but throughout my time at Leeds University.

# Contents

# Chapter 1

# Introduction

## 1.1 Aim

The aim of this project is to reproduce the existing result of a recent paper by Banks and Beason[1] which has shown that it is possible to globally illuminate an isosurface in real-time by precomputing the illumination for all possible isosurfaces within a dataset. In addition, I will describe any problems encountered and set out conditions under which my implementation can be evaluated.

We can split the implementation into two distinct stages. The first stage deals with the precomputation of the illumination, whilst the second involves rendering an isosurface from a volumetric dataset and texture-mapping the illumination generated in the precomputation stage. This distinction between the two phases is important because it is this trade-off of having a precomputation stage that allows for eventual real-time rendering of any chosen isosurface with global illumination applied.

The distinction between the two phases of computation will be subtle in my implementation. This is due to the research nature of the project as it is more convenient to merge the two into one coherent program during development and testing.

The goal of my implementation is to demonstrate real-time interaction (greater than 24 frames per second) with a globally illuminated isosurface and hence independantly verify the result of Banks' paper.

## 1.2 Objectives

The objectives of the project are to:

- Become familiar with the concepts and programming tools for graphics programming. This will involve acquiring fluency with OpenGL, a suitable programming language and development environment.
- Build a system that implements the Marching Cubes algorithm and simple local illumination models allowing realtime interaction with a variety of scalar datasets.
- Research the rendering equation and physical lighting models, using this knowledge to build a simple isometric ray tracer. The ray tracer must be extended to not only handle primitive shape intersection, but also intersection with the implicit isosurfaces within a scalar dataset.
- Use the ray tracer to generate a volumetric texture of illumination values, thereby storing the lighting for a dataset.
- Implement 4D light transport to decouple illumination sampling from isosurface extraction and explore the effects of different sampling methods.
- Evaluate the effectiveness of this method for precomputation and explore its limits.

## 1.3 Minimum Requirements

The minimum requirements are:

- Produce software that compiles and runs on Mac OS X 10.6 to interactively visualise 3D scalar datasets by generating an isosurface.
- Modify the above program to take a volumetric texture that can be used to light an isosurface.
- Generate illumination grids with radiance samples confined to level sets.
- Implement 4D light transport to decouple illumination from isosurface generation and enable uniform radiance sampling.

The possible extensions are:

- Explore the use of different distance measures and exponents to optimise the interpolation function that converts illumination samples into a texture.

- Explore different sampling methods and how the size of the illumination grid can affect the result.

- Look at the possibility of extending this method to volume rendering. It may be possible to precompute the global illumination for a volume rendering, however the viewpoint-dependent lighting associated with this presents questions of how the illumination should be stored.

## 1.4 Milestones

- Create a program that can import a variety of volumetric datasets, extract the dimensions and construct an array of the scalar values.

- Construct a suitable interface with which to display and interact with any visualisations.

- Visualise the volumetric data as isosurfaces by implementing the Marching Cubes algorithm to extract polygonal approximations of the contours contained within the volumes.

- Implement local illumination (Gourad Shading) and smooth the normals to get a baseline against which global illumination can be compared

- Implement an isometric ray tracer that is capable of intersecting some simple geometric primitives.

- Extend the ray tracer handle intersections with the implicit isosurfaces within each dataset.

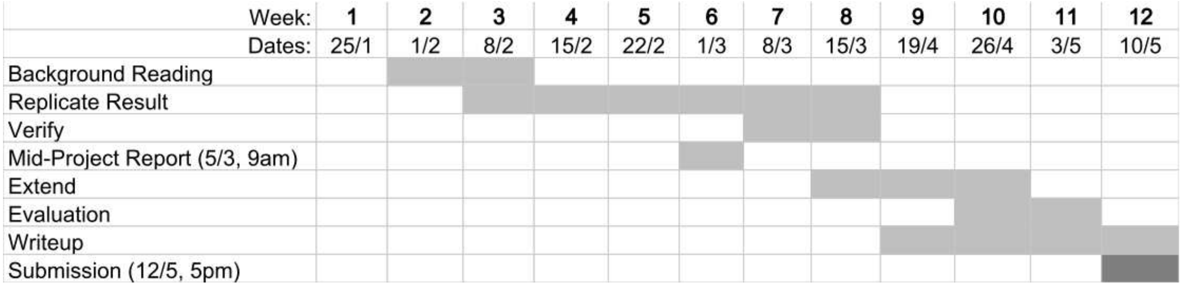
- Generate radiance samples using the ray tracer and visualise these samples as a scattered volume of points.

- Resample the radiance samples as texels for use as a texture.

- Map the texture onto an isosurface.

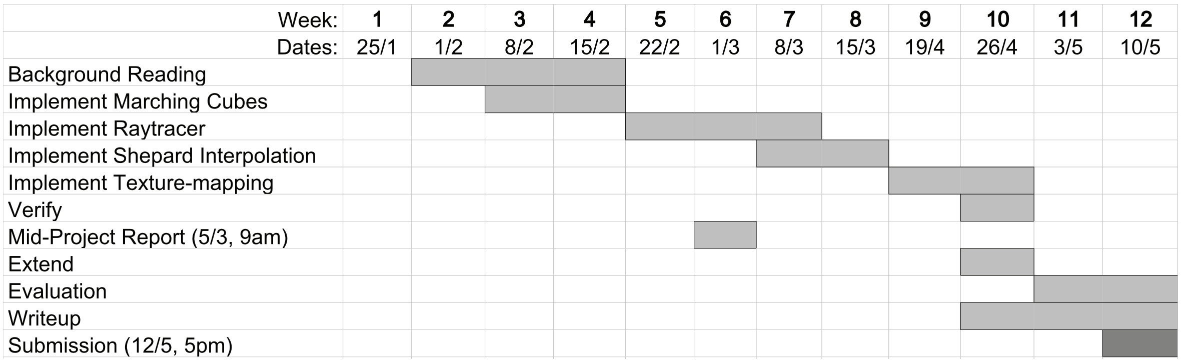
## 1.5 Schedule

### 1.5.1 Original



| Week: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dates: | 25/1 | 1/2 | 8/2 | 15/2 | 22/2 | 1/3 | 8/3 | 15/3 | 19/4 | 26/4 | 3/5 | 10/5 |
| Background Reading | | ▨ | ▨ | | | | | | | | | |
| Replicate Result | | | ▨ | ▨ | ▨ | ▨ | ▨ | ▨ | | | | |
| Verify | | | | | | | ▨ | ▨ | | | | |
| Mid-Project Report (5/3, 9am) | | | | | | ▨ | | | | | | |
| Extend | | | | | | | | ▨ | ▨ | ▨ | | |
| Evaluation | | | | | | | | | | ▨ | ▨ | |
| Writeup | | | | | | | | | ▨ | ▨ | ▨ | ▨ |
| Submission (12/5, 5pm) | | | | | | | | | | | | ■ |

### 1.5.2 Eventual



| Week: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Dates: | 25/1 | 1/2 | 8/2 | 15/2 | 22/2 | 1/3 | 8/3 | 15/3 | 19/4 | 26/4 | 3/5 | 10/5 |
| Background Reading | | ▨ | ▨ | ▨ | | | | | | | | |
| Implement Marching Cubes | | | ▨ | ▨ | | | | | | | | |
| Implement Raytracer | | | | | ▨ | ▨ | ▨ | | | | | |
| Implement Shepard Interpolation | | | | | | | ▨ | ▨ | | | | |
| Implement Texture-mapping | | | | | | | | | ▨ | ▨ | | |
| Verify | | | | | | | | | | ▨ | | |
| Mid-Project Report (5/3, 9am) | | | | | | ▨ | | | | | | |
| Extend | | | | | | | | | | ▨ | | |
| Evaluation | | | | | | | | | | | ▨ | ▨ |
| Writeup | | | | | | | | | | ▨ | ▨ | ▨ |
| Submission (12/5, 5pm) | | | | | | | | | | | | ■ |

The original schedule turned out to unrealistic due to the scale of the project. The topics covered are complex and implementing them correctly took more time and background reading than expected.

# Chapter 2

# Background to the Problem

## 2.1 Overview

In order to explain the implementation of 4D light transport and isosurface illumination, it is first necessary to discuss the nature of computational illumination models and scientific visualisation. This will allow for an analysis of the benefits and limitations to each method as well as the motivations behind new methods.

At the heart of this paper lies the problem of interactively exploring a scientific volumetric dataset that has been lit with global illumination. For the visualisation to be interactive, the implementation must be able to respond to user input and re-render the model within a sufficient amount of time. Current methods for global illumination on commodity hardware take on the order of hours to compute for a typically sized scientific dataset, ruling out the possibility of real-time computation. This means that in order for the rendering to be real-time, we must precompute the illumination and find a suitable method of storage such that it can easily be looked up when needed.

It has been shown that the computation of model geometry can be performed in real-time using the Marching Cubes algorithm. It is also possible to map a 3D texture onto model geometry with negligible impact on performance using OpenGL. With these two facts in mind, if the lighting can be precomputed and stored as a multidimensional texture, then we can simply map the lighting texture onto the model, maintaining real-time interactivity.

This idea of texture-mapped lighting has been demonstrated by a number of researchers including Parker [20], Wyman *et al.* [28], Stewart [24], and Beason *et al.* [2]. All of these existing approaches however suffer from non-uniform sampling of the dataset to generate their lighting texture. This results in an inefficient precomputation phase which does not produce a solution as accurate as could be generated with a more favourable sampling pattern.

To compute the lighting texture using a naïve approach, each extracted surface is illuminated individually thereby coupling the surface extraction step with the illumination step. It is this coupling which results in a non-uniform sampling of the dataset because the samples accumulate along each of the individual extracted surfaces.

Global illumination itself is a vast topic and there are many researchers working on various approaches to improve speed and accuracy. This paper will only cover as much as is necessary to compare and contrast global illumination with local illumination. This project focusses more on precomputation and light transport than with the details of efficient ray tracing. It will be shown that it is possible to precompute the global illumination by implementing only as much as necessary to prove this.

## 2.2 Isosurface Visualisation

Scientist in the fields of medical imaging and computational fluid dynamics (CFD) often need to visualise large volumetric scalar datasets generated from CT scans or computer simulations [12, 16].

The visualisation of volumetric datasets is often achieved by extracting and rendering a surface. Each of these surfaces is defined by a single parameter called an *isovalue*. These surfaces, often referred to as *isosurfaces*, approximate the contours within the data by joining the level sets of specific isovalues. Each isosurface forms a visual representation of a portion of the data.

An isosurface is simply a 3D contour and can reveal much about the underlying features of a dataset e.g. the details of a skull beneath a head. Unlike 2D contours however, drawing multiple isosurfaces is not usually helpful, as one contour can obscure others. Instead, users are given interactive tools and are expected to explore the data by selecting various isovalues.

Isosurface extraction methods often create a geometric representation of the surface by combining geometric primitives such as triangles to form a continuous mesh in 3D space. One such method is the Marching Cubes algorithm.

### 2.2.1 The Marching Cubes Algorithm

The Marching Cubes algorithm [16] introduced by Lorensen in 1987 is typically used to generate an isosurface by creating an explicit polygonal representation. This method marches through the dataset taking each set of 8 adjacent vertices (together, a voxel or cell) and determines which edges are crossed by the surface. The edge points are then interpolated to find out exactly where the contour crosses. With these interpolated points, a set of triangles can be drawn such that when combined, together form a complete surface.

The algorithm requires a lookup table of all possible triangle combinations to find out which edges are crossed and what triangles are necessary. In total there are 256 ($2^8$) possible combinations, although this reduces to 15 when we only count the topologically unique cases. Of the 15 cases, 6 are ambiguous, which can lead to holes in the generated surface. Matveyev discussed a number of solutions to this problem [18], including the use of a lookup table where each case has been chosen using a consistent method.

The Marching Cubes algorithm is capable of extracting isosurfaces in real-time but its performance is heavily dependant on the resolution of the dataset and the number of triangles that are needed to approximate any chosen isosurface. There are many papers on the subject of Marching Cubes acceleration looking at improvements both algorithmically and using high performance hardware [10, 21]. Newman [19] undertook a rigorous survey of the algorithm and looked at various extensions and limitations. His paper summarises the extensive body of publications related to optimisation of the algorithm.

### 2.2.2 Alternatives

Whilst Marching Cubes is the most commonly used algorithm for extracting isosurfaces, there is another method based on tetrahedra rather than cubes that resolves the ambiguity associated with the former. This algorithm, Marching Tetrahedra, works in a similar manner to Marching Cubes in that it marches through the grid of cells, creating triangles in the process. It differs in that each cell is broken down into six tetrahedra [3]. The surfaces created by Marching Tetrahedra are smoother than those created with Marching Cubes, although this comes at the cost of greater computational complexity and an increased number of triangles.

An alternative to extracting isosurfaces is volume rendering (illustrated in figure 2.1). This is a family of techniques also commonly used in scientific visualisation and does not require fitting geometric primitives to the data. Instead volume renderers create images by directly displaying the sampled volume, typically with colour and opacity assigned to each voxel [15]. The primary advantage of volume rendering is the superior image quality. This quality however comes at the cost of additional processing power.

## 2.3 Illuminating 3D Models

When rendering computer generated images of 3D scenes, the question that inevitably arises is of how to shade the rendered objects. Rather than arbitrarily shading the objects, it is possible to place lights within the scene and simulate realistic lighting. This can be done by shading each object according to the placement and properties of the light sources. All methods that simulate lighting in this way can be described as either “local” or “global”. This distinction between local and global depends on

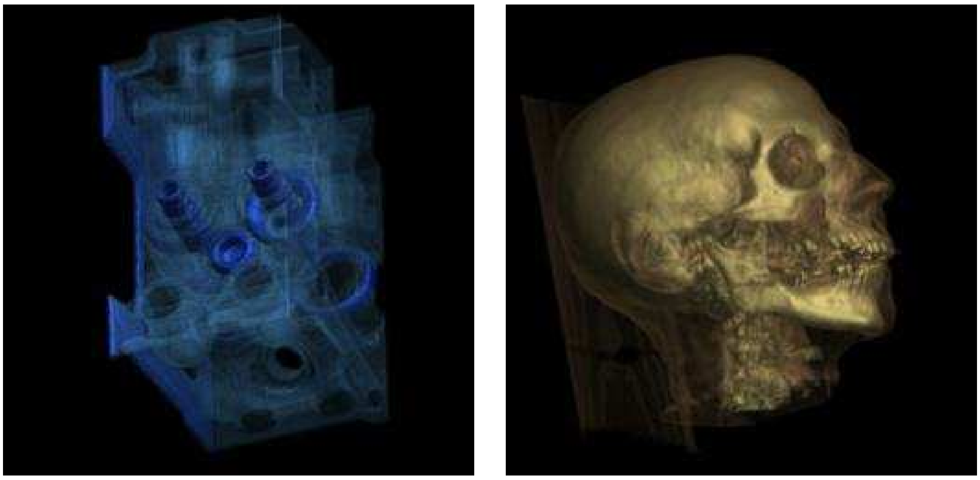


Figure 2.1: **Sample volume rendered images.** These images were rendered using the VolPack library by Stanford Computer Graphics Laboratory with semi-transparent surfaces [14].

the assumptions made in each method. Local illumination is commonly used in real-time graphics but trades realism for computational efficiency by crudely approximating the actual behaviour of light.

### 2.3.1 Local Illumination

The distinguishing feature of local illumination is that each object in a scene is lit individually without regard for any surrounding objects. The assumption made is that light travels from a source to the viewing place having only been reflected off of one object. Whilst computationally simpler, this type of illumination is unable to accurately model any global effects of light such as smooth shadows cast upon one object by another. These global effects occur because in reality, light transport is not limited to such simple paths.

### 2.3.2 Phong Shading Model

Phong shading is one of the most widely used local methods for lighting computer generated models. Phong shading is a general term used to refer to the set of techniques introduced by Phong in 1975 including Phong reflection and Phong interpolation.

Phong reflection is a model for local illumination that describes the shading of a surface as a combination of ambient, diffuse and specular components. Each surface can have different characteristics which we model with constants $k_a$, $k_d$ and $k_s$. These constants specify the ratio of light reflected from the surface for ambient, diffuse and specular lighting respectively. By adjusting these constants, it is possible to model different types of surface such as concrete or shiny plastic.

Individual lights must also be defined by specifying the intensity ($i_a$, $i_d$ and $i_s$), and direction (stored in vector $L$). The Phong reflection model combines the lighting and surface parameters to compute the lighting at each point $p$ on the surface thus:

$$I_p = k_a i_a + \sum_{\text{m} \in \text{lights}} (k_d (L_m \cdot N) i_d + k_s (R_m \cdot V)^{\alpha} i_s) \tag{2.1}$$

where:

$I_p$ is the illumination at a point $p$

$N$ is the normal to the surface at point $p$

$V$ is the direction between point $p$ and the observer

$R$ is the direction of reflection about the normal $N$

$\alpha$ adjusts the size of the specular highlight

It is important to note that only the specular component is reliant on the position of the viewing plane ($V$). Any lighting solution that directly takes into account the position of the observer breaks rotational invariance. This becomes an issue when precomputing lighting and will be discussed later on in chapter 3 when discussing the design choices.

### 2.3.3 Global Illumination

In contrast to local illumination, global illumination methods light a scene as a whole. By more closely modelling the physical properties of light transport, it is possible to generate images of much higher quality.

Global illumination methods such as ray tracing and photon mapping have traditionally been confined to use in feature films and artistic renderings. The reason for this is the computational cost of generating such high quality illumination [26]. With increases in available computing power and the emergence of techniques to precompute high quality illumination [1, 28], real-time globally illuminated rendering using consumer hardware has come within the realms of possibility.

This presents a wonderful opportunity for scientific and medical researchers among others who currently use local illumination methods (typically the default provided by the computer's graphics card [1]) to visualise their datasets. Whilst the more advanced local illumination methods are relatively good for simple models, they are easily surpassed by global illumination when complex models are lit. Depth and proximity perception is vastly improved over local illumination [27] and realism improved by accurately modelling lighting characteristics such as diffuse inter-reflection (colour bleeding), caustics (the envelope of light rays reflected or refracted by a curved object) and penumbra (soft shadows).

## 2.4 The Rendering Equation

The most realistic lighting solution possible for the purpose of generating computer graphics is formulated using the rendering equation. This is an integral equation simultaneously introduced to the graphics community by Kajiya [11] and Immel *et al.* [6] in 1986 which describes how light moves through a scene. Since it is based on the physical laws of how light works, it is the closest model we have for generating illumination and is the standard approach when attempting to globally illuminate a scene. Put simply, this equation describes the light exiting from a point as the sum over a fraction of the light entering the point (direct and indirect) and the total light emitted from the surface at that point. Every global illumination method attempts to solve this equation differently, each tailored to a specific use case but all involve evaluating expensive ray casting functions since each outgoing scattered ray needs to be tested for intersection against every object in the scene. The scattering of rays is computed using the Bidirectional Reflectance Distribution Function (BRDF), which determines the material characteristics of each surface. The rendering equation can be expressed as:

$$L_o(\mathbf{x},\vec{\omega}) = L_e(\mathbf{x},\vec{\omega}) + \int_\Omega f(\mathbf{x},\vec{\omega}',\vec{\omega})L_i(\mathbf{x},\vec{\omega}')(\mathbf{N}\cdot\vec{\omega}')d\vec{\omega}' \tag{2.2}$$

where:

$\mathbf{x}$ is an arbitrary point on a surface

$L_o(\mathbf{x},\vec{\omega})$ is the radiance leaving a point $\mathbf{x}$ in direction $\vec{\omega}$

$L_e(\mathbf{x},\vec{\omega})$ is the emittance at point $\mathbf{x}$

$\int_\Omega \cdots d\vec{\omega}'$ is an integral over the hemisphere surrounding $\mathbf{x}$

$f(\mathbf{x},\vec{\omega}',\vec{\omega})$ is the BRDF, the intensity of light reflected from $\vec{\omega}'$ to $\vec{\omega}$ at point $\mathbf{x}$

$L_i(\mathbf{x},\vec{\omega}')$ is the radiance incident at $\mathbf{x}$ from direction $\vec{\omega}'$

$(\mathbf{N}\cdot\vec{\omega}')$ accounts for the attenuation of incoming light with incident angle respective to normal $\mathbf{N}$

Notice the similarity of the rendering equation to the Phong reflection model expressed in equation 2.1. The rendering equation replaces the sum over all light sources with an integral. This is because the rendering equation must consider the energy emitted by a light source over a hemisphere unlike the Phong reflectance model which treat lights as point sources.

We can rewrite the rendering equation such that the light leaving a point $\mathbf{x}$ is expressed in terms of the incident light at another point $\mathbf{x}'$ from the reverse direction:

$$L_o(\mathbf{x},\vec{\omega}) = L_i(\mathbf{x}',-\vec{\omega}) \tag{2.3}$$

Note that since the incident light now appears on both sides of the equation, we can see that the equation must be evaluated recursively. This recursive property leads to large computational complexity and makes it impractical for real-time evaluation using consumer hardware.

## 2.5 Bidirectional Reflectance Distribution Function

The fraction of light reflected at a point varies across different materials hence we need a function that defines the scattering properties of a surface. This function should take the input and output directions of a ray and return the proportion of light reflected. One such function is the BRDF.

The BRDF describes the reflectance of light from the incoming direction $\vec{\omega}'$ to the outgoing direction $\vec{\omega}$, both of which are defined with respect to the surface normal $\mathbf{N}$ at $\mathbf{x}$. By changing the BRDF, it is possible to simulate a variety of surfaces. Common models include Lambertian reflectance, Phong reflectance and the Oren–Nayar model which simulate perfectly diffuse, plastic-like and rough surfaces respectively. Because Lambertian reflectance only takes into account incident light [8], it differs from most other models in that it is rotationally independent i.e. the lighting it produces does not change as the position of the viewer rotates with respect to the surface. This is important when precomputing illumination because it allows us to ignore the position of the viewer and hence store the result using a smaller amount of space.

## 2.6 Flattened Light Transport

For our purposes of computing the illumination of a volumetric dataset, naïvely evaluating the lighting of a surface in $\mathbb{R}^4$ produces an incorrect result.

When lighting the level sets within a function $h_n : \mathbb{R}^n \rightarrow \mathbb{R}$ in $\mathbb{R}^{n+1}$ there is a subtle problem that arises in which light incident upon one level set can reflect onto another. This problem of light 'leaking' between level sets manifests itself as softer shadows and reduced colour bleeding in the resulting illumination [1]. The solution to this problem lies in confining the light to its originating $\mathbb{R}^{n-1}$ subspace of $\mathbb{R}^n$.

Heckbert's 1992 Paper "Radiosity in flatland" set out one such method for doing so. He described the process for for globally illuminating an isocurve within a 2D plane [9]. This concept of confining light transport to a lower dimension can be extended to lighting an isosurface in $\mathbb{R}^4$ by flattening the light transport to a 3D volume.

Banks and Beason [1] expanded on this idea and introduced a 'flattened' rendering equation (equation 2.4). The purpose this modification was to enable precomputioned illumination of a volume without the need to explicitly construct polygonal representations of individual level sets.

$$L_o^\flat(\mathbf{x},\vec{\theta}) = L_e^\flat(\mathbf{x},\vec{\theta}) + \int_\Theta f^\flat(\mathbf{x},\vec{\theta}',\vec{\theta})L_i^\flat(\mathbf{x},\vec{\theta}')(\mathbf{N}^\flat \cdot \vec{\theta}')d\vec{\theta}' \tag{2.4}$$

By flattening the emitted radiance, emittance, BRDF and the incident radiance along with the hemisphere at $\mathbf{x}$, this equation "forces the graph of the scalar function $h_n : \mathbb{R}^n \to \mathbb{R}$ to be illuminated within $n$-dimensional hyperplanes, just as if the level sets were constructed down in the domain." By illuminating in only $n$-dimensions, the problem of light 'leaking' from one isosurface to another is resolved.

The flattening of light transport for our illumination of $h_3 : \mathbb{R}^3 \to \mathbb{R}$ is convenient because a regular ray tracer operates in $\mathbb{R}^3$. The ray tracing process can therefore be trivially modified for the purposes of generating an regular grid of illumination values. The modifications must ensure that rays are only emitted within a 3D subspace of $\mathbb{R}^4$. Any relections must also be confined to the same 3D subspace.

## 2.7 Precomputation

When viewing volumetric datasets as isosurfaces, the important interactions are rotation of the model and changing of the isovalue. It is these two variables that allow the user to explore the topographic details of the data. Lighting parameters are not typically interactive hence there is no need to dynamically light each dataset at runtime. We can exploit this by precomputing the lighting for a given dataset and applying this cached lighting to the model when viewing.

One method for performing precomputation described by Wyman *et al.* [28] is to extract an isosurface from the data and then globally illuminate it using a standard method such as a ray tracer. This extraction and lighting process is repeated until a sufficient number of illumination samples have been collected. Wyman experimented with various sample densities and concluded that around 100 samples per texel gave acceptable results. Variations in sample size will be discussed in the context of implementation in chapter 4 and evaluated in chapter 5.

The illumination intensity values calculated only ever vary over the 3 spacial dimensions of the model. This allows for the illumination to be stored in a 3D texture as a volumetric grid of illumination intensity values and looked up at runtime. Because all necessary computation has already taken place, interactive rendering can be achieved with the use of this texture.

The problem with this approach of explicitly extracting surface representations and illuminating them is firstly that this requires a lot of space - each isosurface is stored as a collection of triangles, each of which are represented by 3 vertices of 3 floating-point dimensions. When the datasets are large and many isosurfaces are sampled, these requirements can be prohibitory.

Secondly, by illuminating extracted isosurfaces, the illumination samples are confined to the level sets of the surface and hence non-uniform. When interpolated, these non-uniform samples lead to distinct banding in the final texture which yields inaccurate lighting as shown in the Banks and Beason paper [1].

By decoupling the illumination from isosurface extraction, we could achieve uniform sampling and thus more accurate textures without the higher computation and storage requirements of Wyman's method.

# Chapter 3

# Design Choices

Throughout the course of this project it has been necessary to make a variety of decisions and assumptions about how to create an implementation. A number of details were left out the paper by Banks and Beason so it has been necessary to experiment with plausible solutions.

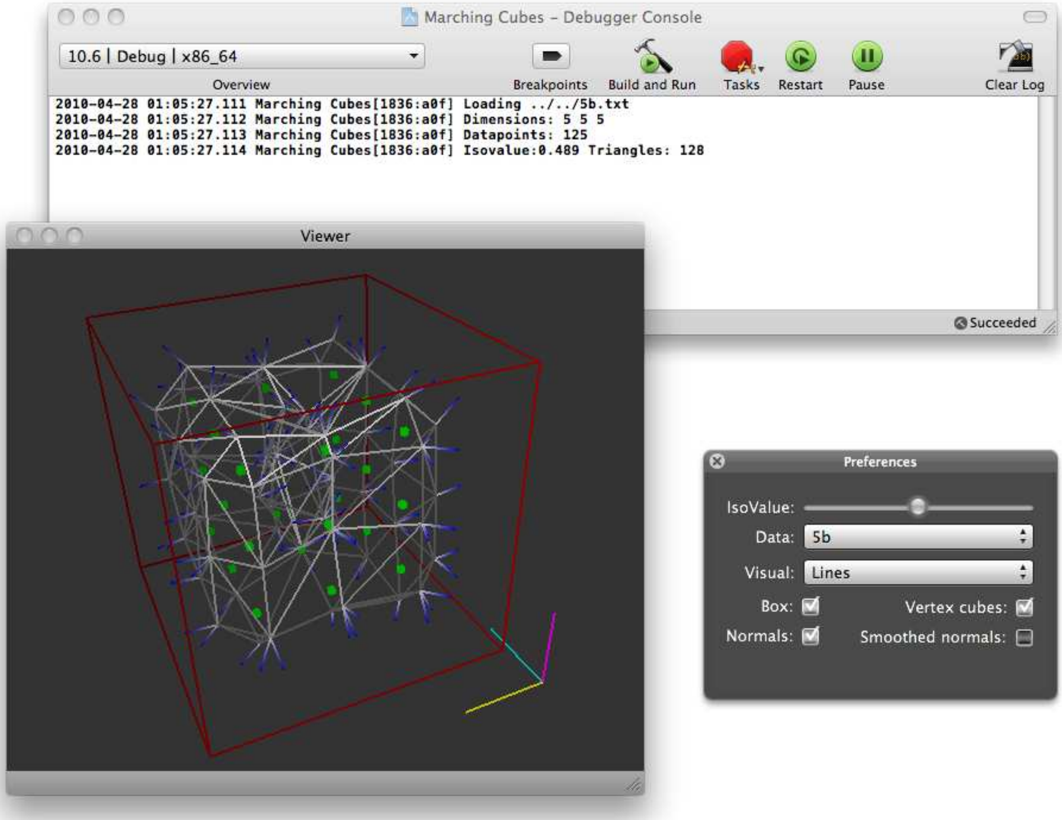
Due to the time constrains of this project, it has also been necessary to decide what areas not to explore. In these cases, the decision has been to focus on the core theme of the Banks and Beason paper which is the novel method of light transport they introduced.

## 3.1 Interface

In order to satisfy the aim of this project it has been necessary to design and build a suitable computer program with which to interactively visualise volumetric data and explore the possible illumination methods.

A suitable graphical interface was needed that would facilitate various interaction and debugging features. The screenshot in figure 3.1 shows the main components of the implemented interface.

The most important part of the interface is the viewing window which uses an OpenGL view to display the visualisation in real-time. The screenshot shows a few of the debugging features such as the drawing of normals turned on. The bounding box around the model with the origin marked allows for easy navigation when the isosurface is small and was helpful in debugging positioning errors. Users can manipulate the rendering by panning, zooming, dollying and rotating. All of this manipulation can be carried out by clicking and dragging on the model whilst pressing various modifier keys. By



```
2010-04-28 01:05:27.111 Marching Cubes[1836:a0f] Loading ../../5b.txt
2010-04-28 01:05:27.112 Marching Cubes[1836:a0f] Dimensions: 5 5 5
2010-04-28 01:05:27.113 Marching Cubes[1836:a0f] Datapoints: 125
2010-04-28 01:05:27.114 Marching Cubes[1836:a0f] Isovalue:0.489 Triangles: 128
```

IsoValue:
Data: 5b
Visual: Lines
Box: ☑ Vertex cubes: ☑
Normals: ☑ Smoothed normals: □

Figure 3.1: **The implementation interface.** This screenshot shows three windows (clockwise from left): the OpenGL viewing window, the debugging console, and the preferences pane

allowing the user to directly manipulate the model, interaction is faster more natural. In addition, the viewing window could be made fullscreen to allow for closer inspection of detailed models.

Rotation is achieved using a trackball method. This is a common and intuitive method which translates the 2D motion of the mouse into 3D rotation as if the user were rolling around an imaginary ball enclosing the model. The translation is done by projecting the coordinates of the mouse onto the imaginary ball to get coordinates in 3D space. The initial click defines a vector from the center of the ball to the surface coordinate. When the mouse is dragged around, a second vector is computed. The cross product of these two vectors gives the axis of rotation and the dot product gives the angle of rotation. The axis of rotation and angle can then be used to rotate the model.

The preferences panel of the interface allow the user to manipulate the model. There are a number of sliders, dropdown menus and checkboxes which all affect the model in real-time. By changing the isovalue slider, the isosurface changes to sweep through the level sets within the dataset. The data menu allows new datasets to be loaded and the Visual menu changes which visualisation is displayed. The different types of visualisation implemented will be discussed in section 4.2. Finally, a set of checkboxes toggle the display of extra information, the most useful of which is the normals. The normals have a gradient applied so that their direction can be seen.

All of the visualisations possible using this interface are lazily computed. Whilst this means there is sometimes a delay when switching between them, the program starts faster and no unused computations are performed. Any program in production deployment would need the option to save and load precomputed lighting textures however this was not necessary for development and was omitted.

Now that we have made decisions regarding the interface, the details of actually drawing and shading isosurfaces must be discussed.

## 3.2 Lighting Properties

When storing illumination data as a texture, we are making the assumption that the lighting calculations only vary over the 3 spacial dimensions of our model. In fact, this is not the case for every component of light that can be modelled. When discussing lighting models in chapter 2, it was noted that the specular component was not rotationally invariant. This is because the specular component changes as the viewing angle varies. Unlike the ambient and diffuse components which can be stored as a single colour value at each point in space, the specular component must be stored in a view independent manner. Wyman has explored this by employing spherical harmonics for all non-diffuse surfaces [28].

Kautz [13] introduced the idea of using a low-order spherical harmonic basis to represent view-dependant lighting and it has since been expounded upon by Green [7] for use in the computer games industry.The use of spherical harmonics would require additional storage and precomputation time.

Also, since non-diffuse surfaces are not necessary for scientific visualisation, this technique has been omitted from the implementation.

For similar reasons, Lambertian reflectance has been chosen as the BRDF because of its isotropy. A surface exhibiting Lambertian reflectance scatters light incident upon it in such a manner that the shading is the same regardless of the viewing angle.

## 3.3 Sampling Density

If we are to precompute the lighting calculations for storage in a texture, it is important that a sufficient number of illumination samples are collected. A high sampling density will ensure that all subtleties of scene lighting are captured and accurately represented in the final texture.

Sampling density has a significant effect on the computation time and it is therefore desirable to find an optimal number of samples to use. It is simple to increase the density by selecting more sample points within the dataset.

During development, it was found that around 10000 samples was the minimum number necessary to achieve acceptable results. For the model sizes used, this works out at 10000/(5x5x5) = 80 samples per texel. For comparison, Wyman found that around 100 samples per texel was sufficient for his more complex models.

Next, we must consider the details of the interpolation function that transforms the raw illumination samples into an array of texels that can be used to map the lighting on to a surface.

## 3.4 Scattered Data Interpolation

In order to create a smooth texture from a set of scattered sample points we must find a way to fill in the gaps between the samples. This is accomplished by mixing the values of the samples that surround any given point. This process of interpolation can be performed in a variety of ways depending on the structure of the input data.

Most interpolation methods such as trilinear interpolation require a rectilinear array (regular grid) of data as input, and output a new rectilinear array of different dimensions or proportions. For our purposes, where we must merge sampled points of computed colour this is not the case. The solution is to use a form of scattered data interpolation that inversely weights sample points. The reason for using the inverse distance as a weight is due to the assumption that nearby points should have more influence than distant points. A function based on this assumption is needed such that for every point $\mathbf{x}_o$ in the output, the colour at that point is defined using only the $N$ input samples $\mathbf{x}_i$, each with color $C(\mathbf{x}_i)$.

## 3.5 Shepard Interpolation

Shepard defined an algorithm that allows us to create such a function [22]. His algorithm allows for a number of choices to be made as to how the selection and weighting of samples is done. Shepard's method is the simplest form of inverse distance weighed interpolation. Although alternative methods exist such as Gradient Plane and Quadratic Nodal Functions, their implementation would require significantly more computation. For our purposes, Shepard's method is sufficient and more complex methods would likely be imperceptible in the end result.

To compute the colour at any point using Shepard's method $\mathbf{x}_o$ in our output we use the following equation:

$$\overline{C}(\mathbf{x}_o) = \sum_{i=0}^{N} \frac{w_i(\mathbf{x}_i)}{\sum_{i=0}^{N} w_i(\mathbf{x}_i)} C(\mathbf{x}_i) \tag{3.1}$$

where:

$$w_i(\mathbf{x}_i) = \frac{1}{d(\mathbf{x}_o, \mathbf{x}_i)^p} \tag{3.2}$$

### 3.5.1 Selection of Nearby Points

In his original paper, Shepard suggested using a minimum of four and a maximum of ten nearby points. His implementation was flexible and attempted to average seven data points by selectively adjusting the search radius. His reasoning for these figures was to include samples representative of the variation within each dimension. Since these figures were based on the interpolation of 2D data, I have extrapolated them to account for the additional dimension in my data, obtaining a minimum figure of six and a maximum of fifteen, with an average of eleven.

In my implementation, nearby points are selected by sorting all of the samples based on their distance from any given point. The nearest eleven points can therefore be selected from the front of the list. This brute force approach must be performed for each sample point and as the majority of the sorted sample points are never used, this approach is inefficient.

The selection of points could be accelerated by reducing the search space. This can be done by using space partitioning methods such as octree or kd-tree structures. Whilst they would improve performance, they have been omitted due to time constraints.

Another acceleration method is to bypass the sorting step altogether and iterate over all samples in any order. To eliminate distant samples, those with distances beyond a given radius are assigned a weight of zero. I was able to implement this method towards the end of the project and found it to be much quicker. Due to time constraints and limited testing, this method has not been used for any of the tests in the evaluation chapter.

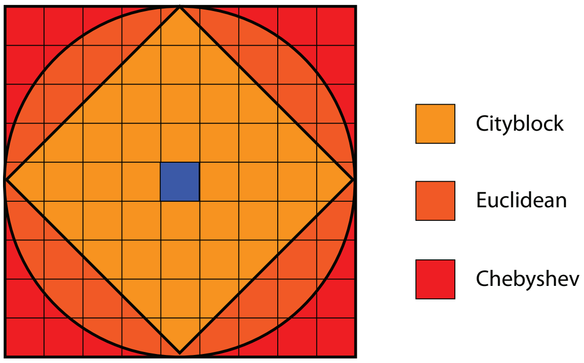


Figure 3.2: **Diagram of various distance measures.** A five unit distance from the center block is compared using Euclidean, Cityblock and Chebyshev distance metrics.

### 3.5.2 Distance Function

It is important to choose a suitable distance function to compute the nearest neighbor queries described above. In the basic brute force case detailed in chapter 4, the input sample set must be sorted for each texel in the output resulting in $f(N) = k \cdot n = O(N)$ computational complexity. Each input sample $\mathbf{x}_i$ must be sorted according to its distance from the output point $\mathbf{x}_o$ which means we must find a suitable distance metric $d(\mathbf{x}_i, \mathbf{x}_o)$.

$$d_{euclidean}(\mathbf{x}_i, \mathbf{x_o}) = \sqrt{\sum_{n=1}^{N} (\mathbf{x}_{in} - \mathbf{x}_{on})^2} \qquad \text{for } N \text{ dimensions} \tag{3.3}$$

$$d_{cityblock}(\mathbf{x}_i, \mathbf{x_o}) = \sum_{n=1}^{N} |\mathbf{x}_{in} - \mathbf{x}_{on}| \qquad \text{for } N \text{ dimensions} \tag{3.4}$$

$$d_{chebyshev}(\mathbf{x}_i, \mathbf{x_o}) = \max_{N}(|\mathbf{x}_{iN} - \mathbf{x}_{oN}|) \qquad \text{for } N \text{ dimensions} \tag{3.5}$$

Using the usual Euclidean distance metric (equation 3.3) involves computing an expensive square root so a more efficient solution is preferable. A cheaper solution is Cityblock distance (equation 3.4) although it is a rather crude approximation. A suitable compromise is to use Chebyshev (chessblock) distance (equation 3.5) which trades slight inaccuracy in the result with improved computation time by eliminating any square roots and using unit distances. In practice, the inaccuracy in the result is likely to be negligible because a number of nearest neighbor samples are chosen and interpolated therefore reducing the dependence of the result of any one specific sample.

## 3.6 Textures

Now that the interpolation of samples has been discussed, the decisions to make regarding textures can be considered. We will first consider the texture size, i.e. the number of texels. A sensible starting size is to have the same number of texels as there are voxels for any given dataset. This is sensible because the lighting resolution then matches that of the given dataset and since the Marching Cubes algorithm generates linear approximations of an isosurface within a cell, it is reasonable to assume that a linear change in colour will provide an acceptable approximation of the lighting.

Wyman experimented with a variety of texture sizes. His findings were that although more dense sampling can give more accurate results in regions where isosurfaces vary significantly, the use of a texture with similar resolution to the original data suffices.

Wyman made the assumption that global illumination generally changes slowly for varying spacial locations. He observed that ”This obviously breaks down with hard shadows and leads to faint shadows where there is no apparent occluder”, but was able to correct this with slight smoothing of the normals over a 4x4 area. Since Wyman and Beason have used textures of the similar resolution as the volume data with good results, I do not feel that it is necessary to change this.

## 3.7 Ray-Voxel Intersection

In order to calculate the global illumination for our scene, it is necessary to be able to intersect rays with isosurfaces. This is because ray tracing relies on recursively tracing rays from origin until intersection with an object. Since our object in this case is a volume rather than a polygon (as is usually the case with ray tracing), it is necessary to find a method that can intersect any of the possible implicit isosurface within the volume. It is possible to extract an isosurface for each sample and then perform intersection on the polygons of that isosurface, but this method is very costly. An analytic intersection method is preferable.

The intersection of a ray and the implicit isosurface contained within a trilinear cell has been described by Parker *et al.* for the purpose of ray tracing [20]. The paper defines an analytic intersection technique which negates the need to generate any geometry and perform iterative intersection testing.

Each ray is parameterised by a parameter $t$ giving the following form $\vec{a}+t\vec{b}$, where $a$ is the starting position of the ray and $b$ is a direction vector. By redefining the trilinear interpolation equation (usually used to find the isovalue at a given point within a cell) to derive a cubic polynomial in $t$. By computing the coefficients of this cubic polynomial, it is possible to find the exact parameter and hence location at which the ray intersects the isosurface.

The algorithm employed also can also quickly test for existence of an isosurface within a cell before computing the exact location. This allows for quicker voxel traversal as this test can be performed

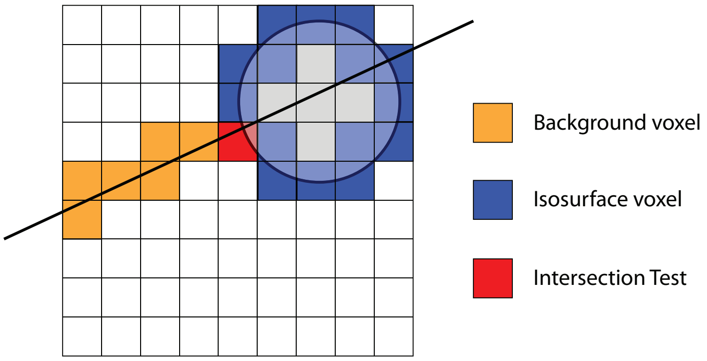


Figure 3.3: **Diagram of voxel traversal and ray-voxel intersection.** The ray (travelling from left-to-right) is discretised as an ordered set of voxels which can be traversed until intersection with an isosurface.

first, and if no isosurface is found then the traversal can continue. Additional work in the area of ray-voxel intersection has been undertaken by Marmitt *et al.* [17]. They discuss a variety of methods for computing fast *and* accurate intersection. Whilst the method described by Parker is accurate, it has not been optimised at all. Marmitt *et al.* improve upon the basic cubic polynomial by first isolating the roots. They make the observation that repeated linear interpolation is fast *if* the starting interval for the iteration contains *only* one root. This is then exploited to gain speedup factors of up to 1.9 over the original algorithm.

Parker *et al.* do not discuss any methods for efficient iteration over all cells. I chose to combine the intersection method described by Parker with an efficient voxel traversal technique described by Šrámek [25]. This technique (illustrated in figure 3.3) reduces each ray to a discrete raster representation of ordered voxels, each pierced by the ray. By performing ray-voxel intersection on each of these ordered voxels in turn, an intersection is found much quicker. Šrámek's technique is an extension of an earlier algorithm developed by Bresenham for the rasterisation of 2D lines [4].

# Chapter 4

# Implementation

## 4.1 Overview

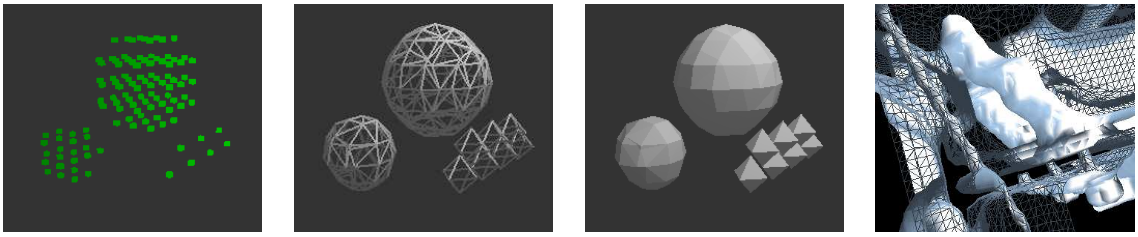
Having discussed design choices and compromises in chapter 3, this chapter will walk through the main stages of the implementation. We will start with the initial visualisation of the data, before moving onto the implementation of flattened light transport and the methods used to sample the illumination of a volume. Since it is necessary to create a texture with which we can map on to an isosurface, the interpolation of samples will be discussed along with the texture-mapping stage.

The first step in implementation is to render the dataset on screen. This involves employing the Marching Cubes algorithm to generate a set of triangles which represent some level set within the dataset.

## 4.2 Visualising the Dataset

As discussed in the background chapter, isosurface visualisation is a popular form of visualisation for isometric datasets and is commonly computed using the Marching Cubes algorithm. By using the Marching Cubes algorithm to generate an isosurface, it is possible to visualise the data in a number of intermediate forms. The first intermediate form is the visualisation of active vertices i.e. those which have a value below the selected isovalue. Using this rendering it is possible to deduce a crude approximation of the form of the data.

Further implementation of the Marching Cubes algorithm produces triangles from which it is



Vertices Mesh Surface Combination

Figure 4.1: **Comparison of different visualisation methods.** The first three images show different visualisations of an atom whilst the final image shows the inside of an engine with the front-side and back-side rendered using different methods.

possible to render a mesh visualisation by drawing lines along the edges of each triangle. This visualisation is more useful than simply showing active vertices as it is much easier to gain an understanding of the topology of the data. An advantage to this method is that it is not necessary to compute any surface normals because the mesh does not need any form of shading.

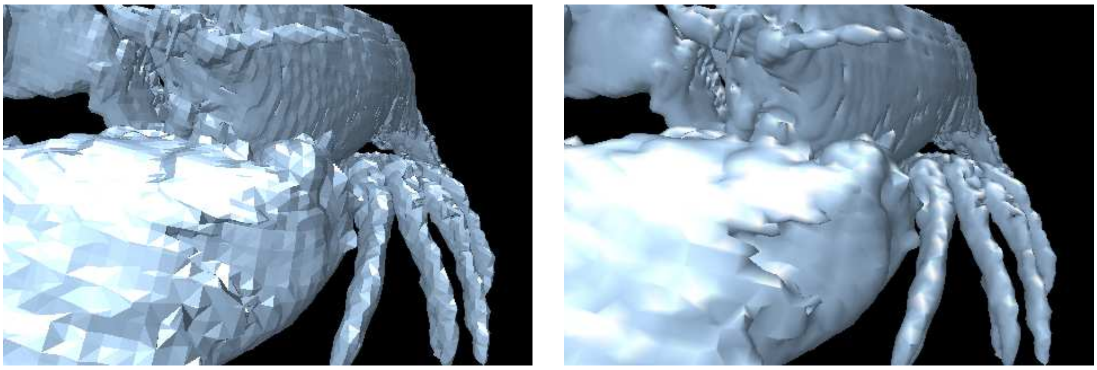
By rendering the triangle from the previous step as solid surfaces and computing the surface normals of each triangle, we end up with a solid shaded surface showing a particular contour within the dataset.

Experimentation with different combinations of the above visualisation methods yields a rendering with which new views of the dataset are possible. The last image in figure 4.1 shows the front-side rendered as a surface with solid shading, and the backside with a mesh.

Simple Gourad shading produces crude discontinuous lighting across the isosurface. By interpolating the normals from the triangles surrounding each vertex we produce vertex normals. Vertex normals van be used to interpolate the illumination across each triangle producing more smoother shading. The images in figure 4.2 show the difference in shading produce by using the original and smoothed normals.

## 4.3 Flattened Light Transport

The next step in the implementation is to implement a global illumination technique in order to compute a more accurate lighting solution. This involves solving the flattened rendering equation described in the background chapter. Although Beason and Banks used a photon mapper, the rendering equation is also typically approximated using ray tracing. Ray tracing is simpler to implements and less expensive to compute so that is the method used here. Since flattened light transport restricts light travelling in the dimension $\mathbb{R}^n$ to $\mathbb{R}^{n-1}$, any ray tracing we do must have the same restriction.



Original

Smoothed

Figure 4.2: **Comparison showing the difference made by using vertex normals.** In this figure, the smoothed normals are used to calculate illumination at the vertices then the resulting colours are interpolated across each triangle in the model.

### 4.3.1 Ray Tracing Mechanism

My implementation of a ray tracer is similar to that described by Shirley [23]. Unlike Shirley's ray tracer which creates 2D images, the one I have implemented is only used for the purpose of sampling the illumination of isosurfaces within a volume. This means that we can make certain simplifications. Firstly, there is no need to simulate a camera and any associated perspective transformations. There is also no need to implement any implicit geometry intersection as the rays we create will only ever intersect an isosurface which has been defined parametrically.

The basis of the ray tracer is the `Ray` class (shown in figure 4.3) which will represent and define the behaviour for all of the rays within our program. Recall from section 3.7 that each ray is parameterised in the form $\vec{a}+t\vec{b}$. This translates directly into the code with $\vec{a}$ represented by the `origin` attribute, and $\vec{b}$ represented by the `direction` attribute. the `pointAtParameter:` method defined for our ray class can now compute any position of the ray given in terms of a parameter $t$.

With the `Ray` class defined, we can now instantiate a set of rays and start computing their intersections. Since we are not taking into account the position of the viewer (as discussed in the design choices chapter), there is no need to start tracing rays from any particular viewpoint. In fact, because we desire a uniform distribution of samples throughout our volume, we can simply generate a set of random coordinates within the dataset bounds and begin the rendering equation.

For each random coordinate, we instantiate a ray and calculate the normal to the implicit isosurface at that point using a method defined by Parker [20]. Using trilinear interpolation, we can also compute the isovalue at this point. The origin of the new ray is set to the chosen coordinate, and the direction and isovalue attributes are set to the newly calculated normal and isovalue respectively.

Using this new ray, we can now begin the recursive process of calculating the radiance incident upon the model at this point as defined by the rendering equation. Note that each ray is operating

```
@interface Ray : NSObject {
    Vec3 *origin;
    Vec3 *direction;
}

@property (assign, readwrite) Vec3 *origin;
@property (assign, readwrite) Vec3 *direction;

- (id)initWithOrigin:(Vec3 *)newOrigin direction:(Vec3 *)newDirection;
- (Vec3 *)pointAtParamater:(float)t;

@end
```

Figure 4.3: **Ray class interface file.** The class has two attributes: origin and direction, each stored as a 3 dimensional vector.

```
[dataset intersectRay:ray withVoxel:current isovalue:val
                                tin:[ray pointAtParamater:tParam]
                               tout:[ray pointAtParamater:tParamNext]]
```

Figure 4.4: **Testing for intersection of a ray with a specific voxel.**

within $\mathbb{R}^4$ because there are three spacial dimensions and an isovalue governing its position. As per the restrictions of flattened light transport, each ray must be confined to the $\mathbb{R}^3$ subspace in which it was emitted. This means that the isovalue associated with each ray never changes during intersection and reflection, preserving the integrity of the ray tracing calculations.

Once the recursive process of computing ray intersections has begun, we need a method for performing the intersection tests on all voxels within the dataset. The dataset is held within its own class and has a method defined for computing ray intersections that handles this process.

Stepping through every voxel in turn would be inefficient as each ray only ever passes through a small number of voxels. Since we know the origin and direction of every ray, we can step through a small ordered subset of voxels using a 3D version of the Bresenham line algorithm, as described by Šrámek. Each voxel is then tested by calling the ray intersection method as in figure 4.4. The `tin:` and `tout:` arguments allow a quick test to be performed to check for the existence of an isosurface as was discussed in chapter 3. Since these parameters are already calculated as a side effect of the Bresenham line algorithm, no extra computation need be performed.

If and when a ray is found to have intersected the volume, the dataset returns a `Hit` object which contains a location, colour, normal and isovalue. We can store this data as an illumination sample. We can also choose to continue the ray tracing process by firing new rays from the intersection point in the hemisphere centered about the normal. Monte Carlo integration can be used to find a set of direction in which the fire new rays.

```
// assume num_samples is a perfect square
sqrt_samples = sqrt(num_samples);

for (int i=0; i<sqrt_samples; i++) {
    for (int j=0; j<sqrt_samples; i++) {
        float x = (float)(i + rand()) / (float)sqrt_samples;
        float y = (float)(j + rand()) / (float)sqrt_samples;
    }
}
```

Figure 4.5: **The code to generate stratified samples.** Adapted from code specified by Shirley.

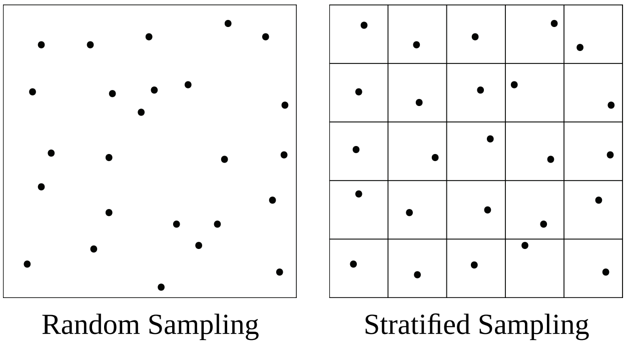


Figure 4.6: **A comparison of random and stratified sampling.** The clustering of samples is evident when random sampling is used. This clustering is mitigated by stratifying the samples.

## 4.4 Sampling Methods

Random sampling as discussed in section 4.3.1 is a common method for picking uniform samples within a given space. It is easy to implement with a random number generator by picking values in the range 0.0 to 1.0 and multiplying by the dimensions of the space to create sample coordinates. The problem with random sampling is that in some unlucky cases, the samples are not uniform and cluster together or leave large portions of the space empty. A solution to this problem is stratified sampling.

This method of sampling works by dividing up the space into a grid of evenly sized subspaces. Each of these subspaces is then filled with randomly positioned samples. By evening out the distribution of samples, there is less likely to be large areas of poor sampling.

Assuming the number of samples has an integer square root, the code to generate stratified samples (shown in figure 4.5) is relatively straightforward.

Shirley observed that stratified sampling in ray tracing almost always produces more visually pleasing results [23].

### 4.4.1 Trilinear Interpolation

Two types of interpolation were used in my program. The first, trilinear interpolation was used to compute the two points at which a ray pierces a voxel. It was also used to find the isovalue of a specified point within a voxel which is necessary when sampling the dataset so that rays could be created.

### 4.4.2 Shepard Interpolation

The other form of interpolation used is Shepard interpolation. As discussed previously, this allows us to resample the radiance samples generated during the ray tracing phase into texels which can be used to texture map the sampled lighting at runtime.

The equation described in section 3.5 translates easily into Objective-C code with the following computation in figure 4.7 being performed for each texel `i`, `j`, `k`. The code shows that the number of samples can be varied, as can the smoothness exponent. The smoothness exponent will be evaluated later on to find a suitable figure.

As discussed earlier, I later found that it was quicker not to sort the samples and instead iterate over them all, assigning a weight of zero to samples outside a specified radius. The modifications to the code necessary to implement this method are trivial. The sorted array is removed and all uses of it are replaced with the unsorted `grid` array. Finally, a test is performed in the weight computation loop which checks for samples outside a given radius and assigns these samples a weighting of 0.

## 4.5 Texture Mapping

The resampled array of texels produced by the Shepard interpolation algorithm must be loaded as an OpenGL texture before we can texture map it onto an isosurface. This is performed by the code in figure 4.9.

The images in figure 4.10 show the relationship between an isosurface and the illumination texture. A false texture has been generated with a dark-to-light gradient to clarify this relationship. Image (a) is an isosurface generated using Marching Cubes then shaded using the Phong lighting model and smoothed vertex normals. Image (b) is a visualisation of the illumination texture which shows a rectilinear grid of colour values centered at the vertexes positions of the original input data. In image (c) the lighting colour values of the lighting texture have been mapped onto the original isosurface. The mapping is done by specifying the coordinates of each triangle vertex (in the range 0.0 to 1.0) as the texture coordinates (also in the range 0 to 1). OpenGL is then able to interpolate the texture values across the isosurface to achieve a smooth gradient, replacing the standard Phong lighting at runtime.

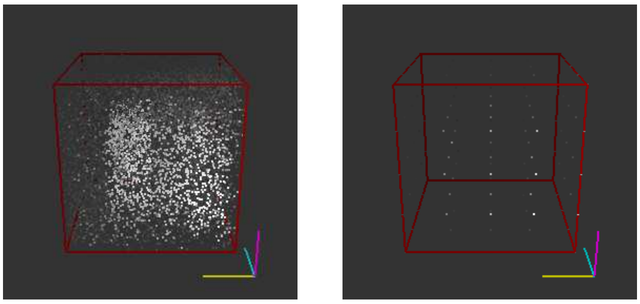
```
// Sort the sample points so that we can choose the closest
Vec3 *texel = [Vec3 withX:i Y:j Z:k];
NSArray *sortedArray = [grid sortedArrayUsingFunction:chebyshevSort
                                              context:texel];

int samples = 11;            // Number of nearest samples to take
float smoothness = 0.5;      // Smooth < 1 < Sharp
float sumOfWeights = 0;
float pointWeights[samples];
Vec3 *localColorSum = [Vec3 withX:0 Y:0 Z:0];
Vec3 *localNormalSum = [Vec3 withX:0 Y:0 Z:0];

// Compute weights of n nearest samples and their total weight
for (int n=0; n<samples; n++) {
    Hit *sample = [sortedArray objectAtIndex:n];
    float d = [current chebyshev:sample.location];
    float pointWeight = 1/pow(d,0.5);
    sumOfWeights += pointWeight;
    pointWeights[n] = pointWeight;
}

// Get the product of the weight and color/normal values then sum all of these
for (int n=0; n<samples; n++) {
    float weight = pointWeights[n] / sumOfWeights;
    [localColorSum add:[[[sortedArray objectAtIndex:n] rgb] product:weight]];
    [localNormalSum add:[[[sortedArray objectAtIndex:n] normal] product:weight]];
}
```

Figure 4.7: **Implementation of the Shepard interpolation calculation.** This is performed for each resampled texel.



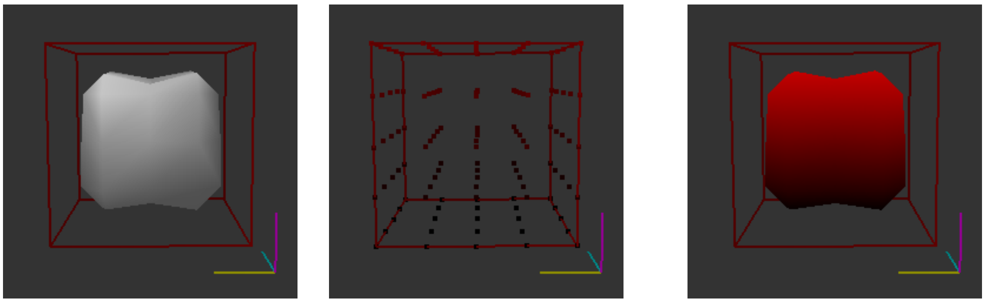
Uniform Radiance Samples        Interpolated Texture

Figure 4.8: **A comparison of the radiance samples and the resulting texture after interpolation.**

```
static GLuint texMap;
glGenTextures(1, &texMap);
glBindTexture(GL_TEXTURE_3D, texMap);

glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_BORDER_COLOR, gr);
glTexImage3D(GL_TEXTURE_3D, 0, GL_RGB, xdim, ydim, zdim,
             0, GL_RGB, GL_FLOAT, image[0][0][0]);
```

Figure 4.9: **The code necessary to load a 3D texture using OpenGL.** The texture has been clamped so that values outside the range 0.0 to 1.0 will not display anything. The texture filtering has also been set to linear so that the texel colours will be interpolated across each rendered polygon. `Image[0][0][0] is a pointer to the resampled output of the Shepard interpolation method.`



(a) Isosurface (b) Illumination Texture (c) Texture-Mapped Isosurface

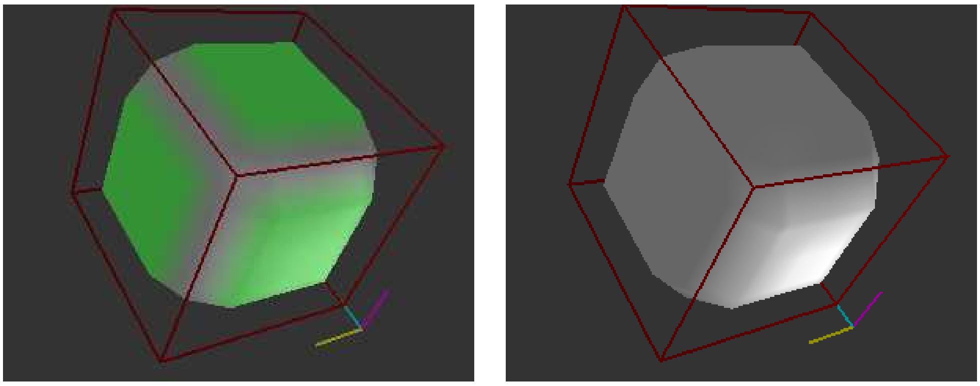
Figure 4.10: **The steps involved in texture mapping.** This figure shows how the isosurface and the illumination texture are of the same dimensions and how the isosurface coordinates can be used to pin parts of the texture onto the surface.

Because the illumination texture can be precomputed, there is no need to perform any heavy computation at runtime. All that is needed is to compute the triangles of the isosurface and then map the texture onto them, both of which it is possible to do in real-time.

### 4.5.1 Mapping Function

The mapping function uses the range 0.0 to 1.0 for convenience because we can treat the texture in a similar manner to the isosurface and perform scaling and rotation transforms without taking into account the original dimensions of the data.

Although it is not necessary for the texture and dataset to be of equal dimensions, the texture will always be of proportional dimensions to the original data so there will be no warping. The means



Border problem

Corrected

Figure 4.11: **An illustration of the texture boundary problem.** In the problematic image, the border colour has been set to green to demonstrate the issue.

that the position the lighting was computed in will be the position it gets mapped to. It would not be necessary to change the mapping function if the size of the texture changed because the projection of the 0.0 to 1.0 coordinates the texel coordinates is handled by OpenGL.

When texture mapping the computed illumination onto an isosurface, there is an issue that can occur at the texture edges. To understand the reason for this we first have to understand how the texture is applied. OpenGL uses texture filtering to create smooth gradients across the triangles of the model. The simplest form of texture filtering is nearest neighbour interpolation. Whilst very fast to compute, this method has crude and ‘blocky’ results. Bilinear filtering is preferable and achieves more accurate results. When using this type of filtering, texture values near the edges (0.0 and 1.0) are a mixture of the border colour and the texel colour. This result in a faint border of constant colour being applied to isosurfaces near the edge of the dataset (shown in figure 4.11). The solution is to stop any interpolation occuring at the edges. OpenGL provides an alternative texture wrapping mode `GL_CLAMP_TO_EDGE` which does not filter edge texels like the standard `GL_CLAMP`.

# Chapter 5

# Evaluation

## 5.1 Technology

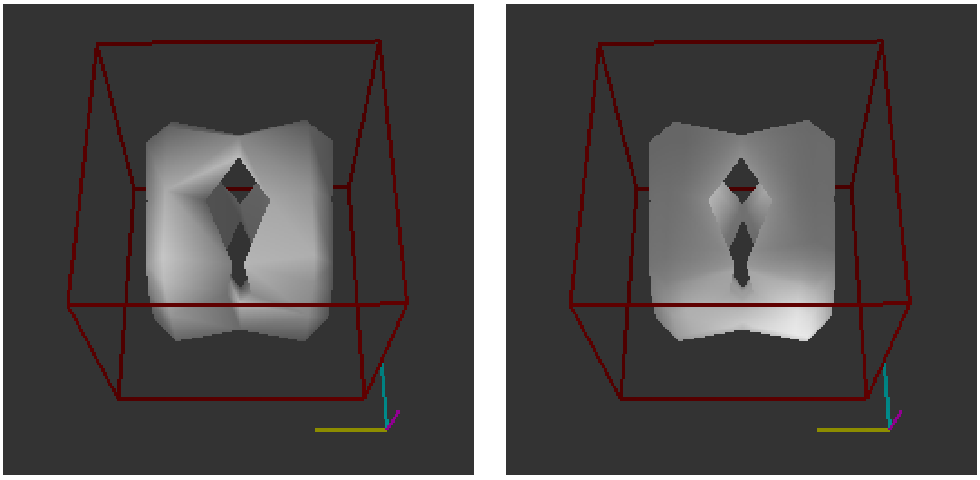
With the performance of the implementation largely dependant on the hardware and software used, it is important that suitable choices were made when evaluating the result.

All of the code produced for this project was written in a combination of Objective-C, C++ and C. As C is a high performance, low level language, critical parts of the program were implemented using this. For other parts of the program such as loading datasets and manipulating the interface, Objective-C and C++ provided acceptable speed and allowed for the use of high level libraries which sped up development. Use of the XCode IDE allowed for integrated design, development and testing.

All tests in this chapter were run on Macbook Pro laptop with a 2.26 GHz Intel Core 2 Duo, 2GB DDR3 Memory and an NVidia GeForce 9400M integrated graphics. As this is my primary machine, its use was convenient and gave consistent results throughout development and testing.

## 5.2 Qualitative Analysis

An evaluation of visual differences is not easily reduced to numbers. To compare the results of this project it is useful to discuss the perceptual benefits. Various authors have looked at perceptual studies using human trials. A recent experiment found that the perception of overall shape as well as relative depth were both improved with the use of physically-based (i.e. global) illumination when compared to local illumination [27]. Another experiment has shown that for at least some surfaces, uniform



Local Global

Figure 5.1: **Comparison of Local and Global illumination.** These images show the difference between a model lit using standard OpenGL techniques and a model lit with the method described and implemented in this paper.

diffuse lighting improves on point lighting in the perception of shape [5]. Stewart, in his paper on vicinity shading called for more studies to be undertaken to verify that global illumination techniques are indeed superior for perception [24].

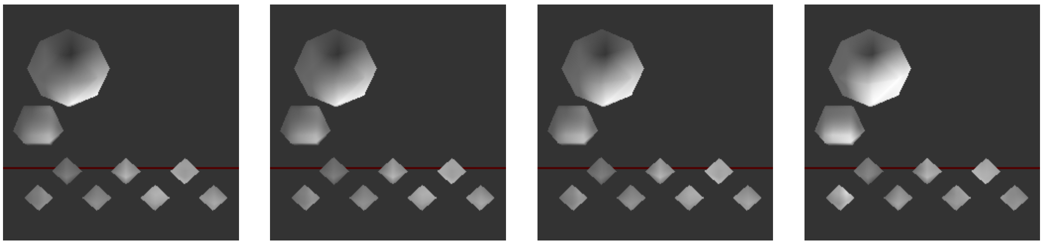
### 5.2.1 Comparison of Local and Global Illumination Results

By comparing the original local illumination with Beason's texture mapped global illumination method, we see a number of visual differences. Bearing in mind the studies mentioned above, if we assume their findings are correct then we can justify the claim that these results are an improvement by looking for evidence of global illumination features.

Although it is difficult to measure the perceptual improvements, the difference in the results are at least self evident. In the locally illuminated image in figure 5.1 we can see that the changes in colour are more pronounced and there is dark shading inside the model. The globally illuminated model is noticeably smoother and we can see that the inside of the model is lighter. We can attribute the smoothness to non linear shading across each polygon and the interpolation used. The lightness inside the global model is most likely due to the diffuse inter-reflection that occurs when accurately modelling the rendering equation.

### 5.2.2 Comparison of Texture Smoothness

As discussed in section 3.5, there is a choice of exponent when implementing Shepard interpolation. By varying this "smoothness" exponent *p*, it is possible to generate textures with different levels of



$p = 0.25$ $p = 0.5$ $p = 1.0$ $p = 2.0$

Figure 5.2: **Comparison of the differences in texture smoothness possible.**

contrast. The images in figure 5.2 show a representative range of the possible results. It is evident that the inverse distance weighting has a more pronounced effect with larger exponents. This is due to sample points further from their corresponding texel being assigned a lower weight. This lower weight reduces the effect of the sample resulting in sharper colour shifts between adjacent texels.

This parameter could be used to fine-tune the interpolation function to suit different styles of model, with lower values of *p* being used for models with less complex topology, and higher values of *p* being used for more complex models requiring more dramatic shifts in colouration and tone. A value of 0.5 was used during development and gave acceptable results across a range of datasets.

## 5.3 Performance

When evaluating the implementation described in chapter 4, there are three main components with which we can measure the performance. The first of these components is the time it takes to precompute the texture of sampled illumination values. The second component is the speed with at which we can generate the geometry of an isosurface. Finally, to evaluate the interactivity of texture-mapped illumination we can compare the performance of isosurface rendering with and without the lighting texture applied.

Performance was tested using the profiling tools within the chosen IDE. The time for each method call was computed and each test was run three times with the final times being the average over all runs. By Comparing individual method times it is possible to find out how different aspects of the computation scale. The profiling tool that was used specified times to the nearest millisecond thus giving a maximum error for each sample of +/- 0.1 ms. Taking into account the propagation of error due to the 3 sample averaging, the standard deviation in the figures below is $\sqrt{3} \approx 1.73$. Where the isovalue has not been specified, this is because it has no effect on computation time.

### 5.3.1 Precomputation Performance

We will evaluate the precomputation time by looking at how it scales to different sized datasets and variations in sampling density. Three datasets have been chosen for their variation in size and com-

| Dataset | 5b | Atom | Engine |
|---|---|---|---|
| Size | 5x5x5 | 13x13x13 | 27x64x64 |
| 100 Samples | 18 | 279 | 30459 |
| 1000 Samples | 217 | 3348 | 335253 |
| 10000 Samples | 3197 | 54465 | 4471965 |
| 100000 Samples | 47961 | 817846 | - |

Table 5.1: **Comparison of total precomputation time (in milliseconds) with varying sample sizes.** Missing results indicate that the computation took an unreasonable amount of time to compute.

| Dataset | 5b | Atom | Engine |
|---|---|---|---|
| Size | 5x5x5 | 13x13x13 | 27x64x64 |
| Shepard Interpolation | 1909 | 33730 | 2789026 |
| Chebyshev Distance | 702 | 12627 | 1032972 |
| Sample Creation | 32 | 32 | 195 |

Table 5.2: **Comparison of computation time (in milliseconds) for each method performed during precomputation.** 10000 samples were used for each test.

plexity.

We can see in table 5.1 that the total time taken to compute the lighting scales linearly with both sample size and dataset size. The missing figure for the Engine dataset is because the test was stopped after two hours.

Table 5.2 clearly shows that it is the Shepard interpolation that takes up the majority of the precomputation (about 70%). It is therefore reasonable to assume that this would be a good starting point for optimisation. The naïve method used to find nearest neighbour samples could be optimised with space partitioning methods or a collection type algorithm rather than a full sample space sort.

### 5.3.2 Isosurface Extraction Performance

A common method of evaluation for real-time graphics is to calculate the rendering framerate (in frames per second). Table 5.4 compares the framerates for models of different sizes with the default local lighting, and with the new texture-mapped global lighting.

The aim of the new method is to achieve more accurate lighting whilst maintaining interactive rendering rates. This means a successful implementation should achieve real-time framerates (greater than 24 fps) for the new method with the trade-off being precomputation time. 24 frames per second has been chosen because this is a common rate at which movies are displayed and would likely fit most peoples definition of real-time.

As discussed in the background chapter, marching cubes performance is heavily dependant on

| Dataset | 5b | Atom | Engine | NegHip |
|---|---|---|---|---|
| Size | 5x5x5 | 13x13x13 | 27x64x64 | 64x64x64 |
| Isovalue | 0.5 | 0.5 | 127 | 127 |
| Triangles | 125 | 152 | 34306 | 16656 |
| Marching Cubes | 60 | 59 | 4 | 2 |

Table 5.3: **Comparison of Marching Cubes computation rate (frames per second).** When recalculating the isosurface and its normals, we see that performance is heavily dependant on the size of the dataset.

| Dataset | 5b | Atom | Engine | NegHip |
|---|---|---|---|---|
| Size | 5x5x5 | 13x13x13 | 27x64x64 | 64x64x64 |
| Isovalue | 0.5 | 0.5 | 127 | 127 |
| Triangles | 125 | 152 | 34306 | 16656 |
| Local illumination | 59 | 60 | 29 | 52 |
| Global illumination | 55 | 59 | 19 | 30 |

Table 5.4: **Comparison of illumination method framerates (frames per second).** The tables shows that texture-mapped illumination has negligible impact on speed for small datasets. On the larger datasets with more triangles, rendering speeds are 57% and 65% as fast as the default method. The rendering speed is more dependant on the number of triangles rather than the size of the dataset as evidenced by the Engine and NegHip datasets.

the size of the dataset. This is seen in the performance figures in table 5.3. The framerates achieved clearly decrease with the increasing size the datasets. Although the NegHip dataset has 51% fewer triangles than the Engine dataset, its performance is twice as slow. The Neghip dataset has 57% more voxels however which explains the increase in computation time.

### 5.3.3 Texture-Mapping Performance

As specified throughout this paper, the aim is to achieve real-time rendering which is defined as greater than 24 frames per second. Looking at the figures from table 5.4, it is evident that this goal has been met for reasonably complex datasets. The NegHip dataset,

Although the figures above show that my implementation of local illumination is faster on larger datasets, Wyman observed that his implementation was actually slower. This is because he used Phong shading with separately computed hard shadows in comparison to my implementation which just computed simple diffuse Gourad shading without any shadows.

### 5.3.4 Summary

In summary, my implementation is able to display texture-mapped global illumination at real-time framerates for small datasets. With further development, there is significant scope for improvement of these figures through the use of optimisation algorithms and by taking advantage of dedicated hardware.

The bottleneck is clearly the isosurface extraction. As was discussed in earlier chapters, this is an active area of research with many hundreds of papers on the topic of Marching Cubes optimisation. With more time, I would like to explore some of the hardware-oriented approaches.

The integrated graphics card use in these tests is not particularly powerful. I would like to see these tests run on a more powerful and better equipped machine because it is likely that more realistic figures would be achieved.

## 5.4 Memory Usage

During the precomputation phase, a potentially large amount of memory is taken up by the sampling and interpolation process. Since we store various values for each sample and there is commonly many thousands of samples being stored, if memory is not properly handled or large datasets are loaded, then available physical memory may become an issue. The datasets I am using should not be a problem. Large, full size datasets may be however.

Assuming `double` is used for storage of all floating point numbers and that the largest dataset used in the evaluation is chosen (64x64x64). For each sample the following attributes are stored:

Colour: 24 bytes
Location: 24 bytes
Normals: 24 bytes
isovalue: 8 bytes
= 80 bytes per sample

The samples are then resampled onto a grid the same size as the dataset:

24 bytes * 262144
= 6291456 bytes

In addition, memory is required to hold the geometry computed at runtime:

Vertex positions: 72 bytes
Normals: 24 bytes
Vertex Normals: 72 bytes
= 168 bytes

And the dataset takes up:

8 bytes * 262144 voxels
= 2097152 bytes

This gives a simple estimate of the memory requirements for my program. Totalling up the values for 10000 samples and 16656 triangles, we get a total of 11.43 MB. In reality, when this dataset is loaded the program uses 118 MB of memory.

Unfortunately I encountered memory issues when computing tests on larger datasets. I found that far more memory than expected was being used when computing some of the larger datasets. The profiling tools within my IDE show that there are various memory leaks in the program. Unfortunately there has not been enough time to find and correct all of these.

# Bibliography

[1] David C. Banks and Kevin Beason. Decoupling illumination from isosurface generation using 4d light transport. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1595–1602, 2009.

[2] Kevin Beason, Josh Grant, David C. Banks, Brad Futch, and M. Yousuff Hussaini. Pre-computed illumination for isosurfaces. volume 6060. SPIE, 2006.

[3] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 1988.

[4] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1), 1965.

[5] Heinrich Blthoff and Michael S. Langer. Perception of shape from shading on a cloudy day. *Journal of the Optical Society of America*, 11(11):467–478, 1999.

[6] Michael Cohen, Donald Greenberg, David Immel, and Philip Brock. An efficient radiosity approach for realistic image synthesis. *IEEE Computer Graphics Applications*, 6(3):26–35, 1986.

[7] Robin Green. Spherical harmonic lighting: The gritty details. Proceedings of Game Developers Conference, 2003.

[8] Donald Hearn and Pauline Baker. *Computer Graphics with OpenGL*. Prentice Hall, 3rd edition.

[9] Paul S. Heckbert. Radiosity in flatland. *Computer Graphics Forum*, 2:181–192, 1992.

[10] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *Proceedings of Center for Advanced Studies on Collaborative Research 2006*, page 39, New York, NY, USA, 2006. ACM.

[11] James T. Kajiya. The rendering equation. *Proceedings of ACM SIGGRAPH*, 20(4):143–150, 1986.

[12] Arie E. Kaufman. *Volume Visualisation in Medicine*. Academic Press, 2000.

[13] Jan Kautz, Peter-Pike Sloan, and John Snyder. Fast, arbitrary brdf shading for low-frequency lighting using spherical harmonics. In *Proceedings of the 13th Eurographics Workshop on Rendering*, pages 291–296, Aire-la-Ville, Switzerland, 2002. Eurographics Association.

[14] Philippe Lacroute. *The VolPack Volume Rendering Library*, 1995 (accessed 11th May, 2010). `http://graphics.stanford.edu/software/volpack/`.

[15] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Computer Graphics*, 9(3):245–261, 1990.

[16] Willian Lorensen and Harvey Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21, 1987.

[17] Gerd Marmitt, Andreas Kleer, Ingo Wald, and Heiko Friedrich. Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *Proceedings of Vision, Modeling, and Visualization*, pages 429–435, 2004.

[18] Sergey V. Matveyev. Approximation of isosurface in the marching cube: Ambiguity problem. In *Proceedings of Visualization 1994*, pages 288–292, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[19] Timothy S. Newman and Hong Yi. A survey of the marching cubes algorithm. *Computers and Graphics*, 30(5):854 – 879, 2006.

[20] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive ray tracing for isosurface rendering. In *Proceedings of Visualization 1998*, pages 233–238, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[21] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Joint Eurographics - IEEE TVCG Symposium on Visualization*, pages 293–300, 2004.

[22] Donald Shepard. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of ACM National Conference*, pages 517–524, New York, NY, USA, 1968. ACM.

[23] Peter Shirley. *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2000.

[24] A. James Stewart. Vicinity shading for enhanced perception of volumetric data. In *Proceedings of IEEE Visualization 2003*, page 47, Washington, DC, USA, 2003. IEEE Computer Society.

[25] Milos Šrámek and Arie E. Kaufman. Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics*, 6(3):236–252, 2000.

[26] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164, 2001.

[27] Chris Weigle and David Banks. A comparison of the perceptual benefits of linear perspective and physically-based illumination for display of dense 3d streamtubes. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1723–1730, 2008.

[28] Chris Wyman, Steven Parker, Peter Shirley, and Charles Hansen. Interactive display of isosurfaces with global illumination. *IEEE Transactions on Visualization and Computer Graphics*, 12(2):186–196, 2006.

# Appendix A

# Personal Reflection

I am satisfied with the outcome of this project, in particular the quality of software I was able to develop and the vast amount of new knowledge I have aquired. The personal goal of learning to program on the Mac has been met as an intentional side effect of this project and has sustained my motivation when the implementation became tough.

This has been an a particularly challenging and intellectually stimulating project for me. Although initially unsure of my abilities to comprehend and implement such topics, I am very pleased that I went with my instinct and chose something that I found interesting – work never seems like work when you enjoy it. Whilst I am frustrated that the project has come to a close, I am also excited at the prospect of continuing the development of my software as there are so many possible directions to take it in. I would advise any students picking a project to go for something slightly beyond their accepted reach as the hard work put in brings great rewards.

Thanks to the regular meetings and contact with my supervisor, I would stress the importance of continuous discussion of progress. Whilst it may not be pleasurable, regular evaluation of progress keeps you from straying to far from the project aims. I found that writing descriptions how components of the project fit together, and then refining these descriptions allowed me to focus my thinking and not not lose track of the logical flow of the project. By consolidating your knowledge, it is easier to find out where the holes are and you are able to better communicate that knowledge to those around you. It all too easy to get caught up in the details of particular algorithms and lose track of the higher ideas of why each part is necessary and how they relate.

Another bit of advice I would give is to do comprehensive research into all areas that the project entails. I found myself continually discovering new knowledge which would have been useful in planning earlier on. This also prevents the scenario where you reinvent the wheel only later to find out it was done by someone else way before you!

The last piece of advice I would give to future students is to take control of the project from the beginning and decide the exact direction in which you wish to go rather than waiting for someone to tell you. This requires you to continuously think about the goals of the project and make sure you are able to meet them. In the end it, is you who must carry out the work and so it must be you who understands where to take it.