Generating Radiosity Maps on the GPU

by

Gabriel Moreno-Fortuny

A thesis
presented to the University of Waterloo
in fultilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Global illumination algorithms are used to render photorealistic images of 3D scenes taking into account both direct lighting from the light source and light reflected from other surfaces in the scene. Algorithms based on computing radiosity were among the first to be used to calculate indirect lighting, although they make assumptions that work only for diffusely reflecting surfaces. The classic radiosity approach divides a scene into multiple patches and generates a linear system of equations which, when solved, gives the values for the radiosity leaving each patch. This process can require extensive calculations and is therefore very slow. An alternative to solving a large system of equations is to use a Monte Carlo method of random sampling. In this approach, a large number of rays are shot from each patch into its surroundings and the irradiance values obtained from these rays are averaged to obtain a close approximation to the real value.

This thesis proposes the use of a Monte Carlo method to generate radiosity texture maps on graphics hardware. By storing the radiosity values in textures, they are immediately available for rendering, making this algorithm useful for interactive implementations. We have built a framework to run this algorithm and using current graphics cards (NV6800 or higher) it is possible to execute it almost interactively for simple scenes and within relatively low times for more complex scenes.

Acknowledgements

Thanks to my supervisor, Mike McCool, who came up with many ideas and helped me in much of the research. Thanks to the two selected readers, Stephen Mann and Peter Forsyth, who gave me valuable feed back and support. Also thanks to the Computer Graphics Lab members who helped me in solving many of the implementation issues. Thanks to CONACYT who funded me. Thanks to my soul-mate, Flor, who supported me through the hardest times. I would like to dedicate this thesis to her and to my parents who gave me all the support I needed to get here in the first place.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

When trying to generate photorealistic images of 3D scenes, there are many aspects of the physics of light that must be taken into account. One particular aspect is the way in which objects affect the general illumination of a scene by reflecting or refracting light and by casting shadows. Global illumination algorithms are those that specifically take this aspect into account to generate photorealistic images.

The problem with calculating the global illumination of a scene is that an energy equilibrium system must be solved. One approach to solve this problem is called *radiosity*, after the units in which it represents its solution. This approach works exclusively for objects with diffusely reflecting surfaces because it is possible to simplify the equations that model the light interactions between these kinds of surfaces. Radiosity is based on finite element discretization, that is, each surface is divided into small patches. For each pair of patches a term called *form factor*, which geometrically relates the two patches, must be computed. With this, a system of equations that relates all patches is generated, taking into account emitted and received light as well as visibility between them.

Solving these systems requires a large amount of processing and can be very problematic for complex scenes. Setting up the system in the first place is a problem because a large number ($O(n^2)$) of form factors are required and computing them all can be even more expensive than actually solving the resulting system of equations. An easier alternative that can give equally good results is to use a stochastic approach called *Monte Carlo* to calculate the geometric relation given by the form factors. With a Monte Carlo approach,

the solution to a problem is given by the expected value of a random variable, such that it is possible to converge to that solution by generating many iterations of the random variable. When using a Monte Carlo based method to solve a radiosity system, a large number of rays with random directions are shot from one patch to another. The geometric relation between the patches can be obtained from the probability that each ray coming from one patch has of hitting each other patch. Using this in conjunction with iterative matrix solving methods, it is possible to compute the complete solution to the system without having to store or explicitly calculate form factors.

This thesis proposes the use a Monte Carlo method that is designed to take advantage of the processing architecture offered by current Graphics Processing Units (GPUs) (for example the NVIDIA GeForce 6800 and 7800). The main idea is to use textures to store the different values required and to use multiple rendering passes to iteratively solve for the final radiosity values. A texture atlas is created in which each texture element (texel) represents a square patch in the scene. Using a GPU based ray tracer, the system can shoot a ray from each of these texel patches and store the value of the radiosity emitted from the surface it hits. Starting from any initial approximation, for instance one computed exclusively from direct lighting, our system will rapidly converge to the correct solution.

This algorithm can be implemented with high efficiency and performance in current GPUs and it is possible to have almost interactive renderings for small scenes. A simple framework based on this approach has been implemented to show the potential of the algorithm. It presents a scene with only direct illumination at first, but then progressively converges to a radiosity solution. This allows the user to move around the scene without having to wait for the complete solution to be obtained. Interaction with objects and light sources in the scene is allowed and when these change, the radiosity solution is gradually recomputed. Though a complete framework for this implementation is presented in this thesis, there are many optimizations that can be added to obtain faster convergence and higher quality images. It should be possible to have real interactive rates for a complete solution in the near future because GPUs are improving rapidly and more optimizations remain to be exploited for future implementations.

Chapter 2 of this thesis will explain all the background regarding the details of the proposed algorithm and also mention some of the previous work done on this field. Chapter 3

will explain the algorithm in detail and show mathematically how it works. Chapter 4 will delve into the programming specifics and discuss the issues regarding GPU programming. Chapter 5 will show some comparisons that were done to prove the validity of the algorithm and will also show rendered examples of various test scenes. Finally, Chapter 6 will summarize the conclusions obtained from this thesis and propose future extensions and possible optimizations.

In addition, at the end of the thesis there are some helpful appendixes. Appendix A has definitions of some of the technical terms used here. Appendix B describes the mathematical notation used in chapters 2 and 3. Lastly, Appendix C contains descriptions and listings of pseudocode for various programs found in Chapter 4.

Chapter 2

Background

This chapter presents a brief history of the development of illumination methods and also explains the mathematics behind the algorithm presented in this thesis. The first section will give an overview of three types of rendering algorithms. The second and third sections explain global illumination and specifically the mathematical background of the radiosity approach. Finally, the last section describes the latest advancements in graphics hardware and how global illumination methods have taken advantage of this.

2.1 Illumination Methods

Illumination methods can be divided into the following three types: *rasterization*, *ray tracing* and *global illumination*. In all three methods a scene description is necessary as input. This description includes many possible attributes for each object or polygon making up the scene, among them vertex positions, normals, colors, and many more. In addition, camera attributes that describe the point of view must also be included and can include view position, view direction, field of view and others. Each of these methods is described in the following subsections.

2.1.1 Rasterization

Rasterization is the standard way to generate images for interactive 3D graphics like the ones used in games and visualization systems. In this method, the input polygon attributes go through a series of transformations and are projected onto a 2D coordinate system with the correct perspective. For each projected polygon visible within a viewing window grid, a series of fragments are generated that correspond to discrete grid samples of each polygon. These fragments are further modified with shading operations that take into account textures and other special effects before resulting in a final pixel color.

This method is efficient because each vertex is processed only once and the fragment processing only applies to those fragments from polygons or sections of polygons that were inside the viewport. In addition, each fragment is processed using only local information, that is, the illumination calculations for a particular fragment do not include the rest of the geometry. Current graphics processors are optimized to execute these operations at extremely fast speeds and have multiple vertex and fragment processing units which allow massive parallelism. However, lighting interactions between objects in a scene (like shadows and reflections) are not easily computed this way, so the images generated often lack realism.

2.1.2 Ray Tracing

In 1980, Turner Whited proposed the ray tracing approach to image generation [34]. Ray tracing involves calculating the result of shooting or tracing a ray from the viewing point into the scene to see what object it hits. Once the point where a surface was hit is known, local surface shading can be performed and additional rays can be traced to achieve effects like shadows, reflections and refractions.

This process is extremely time consuming because every time a ray is shot, it must be compared with every single polygon to see if the ray was hit. To optimize this, many algorithms have been developed that reduce the amount of geometry that must be checked by the ray as it is traced through the scene. For example, one way is to use a grid to divide the scene uniformly into 3D cells (called voxels) so that the ray only checks the geometry within each voxel [11, 13, 30]. Another way is to use a kD-tree based method,

which also divides the scene, but creates more divisions where the scene has more complex geometry [19].

In general, ray tracing can be slow, but it takes into account more of the global physical properties of illumination than rasterization. However, ray tracing on its own does not consider indirect diffuse reflections.

2.1.3 Global Illumination

In real life, surfaces are not only directly illuminated by light sources, but also by light reflected from other surfaces. The idea of global illumination is to take into account this indirect light. In theory, every object in a scene can affect every other, making global illumination a challenging problem.

Ray tracing includes shadows, which affect illumination, and can also take into account reflected and refracted light, so in a way it can be considered a first step towards global illumination. However, there are other relevant global aspects of illumination that can be taken into account: illumination from reflections of diffuse surfaces, caustics (intense light caused by reflective or refractive surfaces that bend and focus light), interactions of light with participating media like gas or fog, bending of light due to heat and polarization, and others.

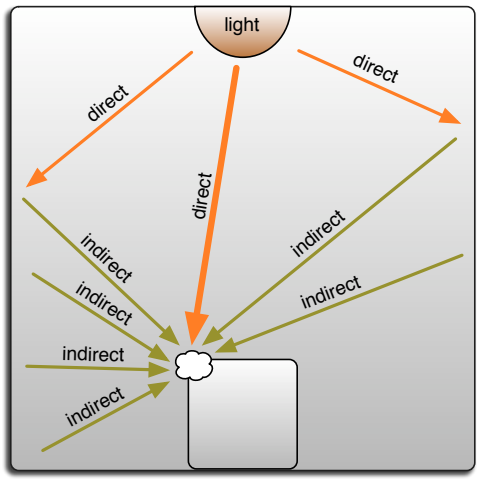


Figure 2.1: Indirect lighting.

Perhaps the most common effect that people relate to global illumination is that of indirect diffuse lighting (see Figure 2.1). The first algorithm used to compute this component of illumination was the radiosity algorithm, a finite element algorithm published by researchers from Cornell University in 1984 [14]. Later, other algorithms were developed to calculate other aspects of global illumination, and to address some of the limitations of the radiosity approach. For example, an extension of ray tracing, *path tracing* [22], recursively shoots a single ray in a random direction after hitting each surface and these rays combine to generate random paths. For each pixel a large number of paths are then followed to find the contribution of light that follows each path. These contributions are averaged and result in an estimate of indirect lighting and can be used for many other effects. Another example is *photon mapping* [20, 21] in which photon particles are shot from the light sources and then a map of the photons' directions and where they hit is generated so that, in a second pass, indirect illumination and caustics can be calculated.

For the method proposed in this document, I make the same simplifying assumptions as the original radiosity method but use a random sampling approach similar to that of path tracing and create maps for additional passes as is done in photo mapping. The following sections detail the mathematics behind the concept of calculating the radiosities of diffuse surfaces.

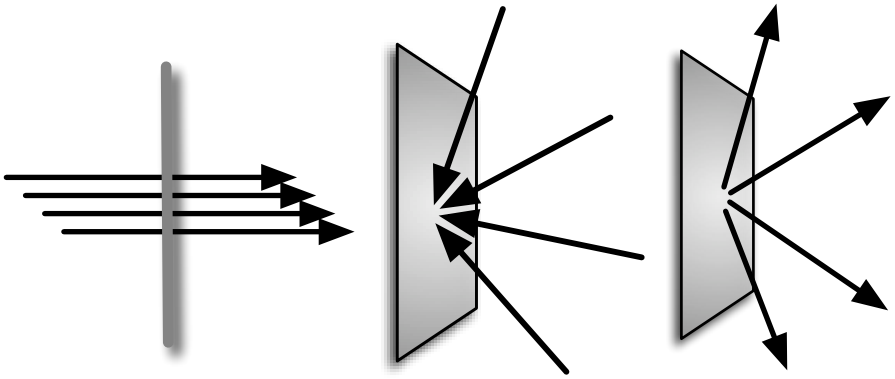
2.2 The Classic Radiosity Approach

To explain how the radiosity of a scene is calculated using the classic finite element method, it is necessary to first describe some concepts from the science of radiometry, the area of study involved with the physical measurements of light. Here is a brief explanation of some of the basic radiometric units, adapted from *Advanced Global Illumination* [9].

2.2.1 Radiometry

Light can be expressed as *radiant power* or *flux*: the total energy (light energy in this case) that flows through a surface per unit of time (see Figure 2.2(a)), measured in Watts (W). Two other useful quantities are *incident radiant power*, also known as *irradiance*, which measures how much total light power arrives on a surface (see Figure 2.2(b)) and *exitant*

radiant power which measures how much light leaves a surface (see Figure 2.2(c)). Exitant radiant power is also known as *radiosity* and it is this concept from where this approach gets its name, since we calculate how much radiosity leaves from each part of a scene. Both irradiance and radiosity are measured in Watts per square meter (W/m^2).



(a) Flux. (b) Irradiance. (c) Radiosity.

Figure 2.2: Radiometric quantities.

The most important measure used in radiometry is *radiance*, defined as flux per unit projected area per unit solid angle. Radiance is a five dimensional quantity: a three dimensional (x, y, z) position (represented in the equations in this document as x) and a two dimensional solid angle (σ) with direction $\vec{\omega}$. It can be pictured as the amount of flux that reaches or leaves a point on a given surface while going through a given area projected on a hemisphere above that point (See Figure 2.3).

Radiance is also dependent on the wavelength of light energy, however, for the mathematical derivations found in this thesis we neglect this[1]. One important property of radiance is that it is independent of the direction in which the flux is flowing. Thanks to this property, it is possible to represent both radiosity and irradiance as an integration of

[1]In computer graphics, three colors are generally used to represent the whole range of light: red, green and blue (RGB). Since all the computations are the same for the three color elements so the mathematical derivations will be made assuming just a monochromatic color, that is, with only one albedo (ρ) value.

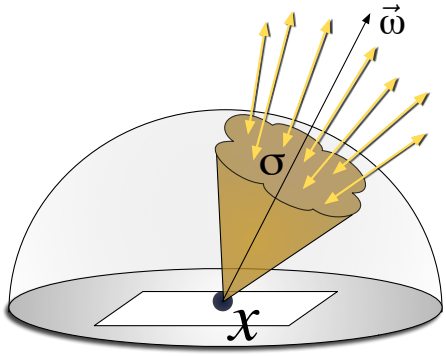
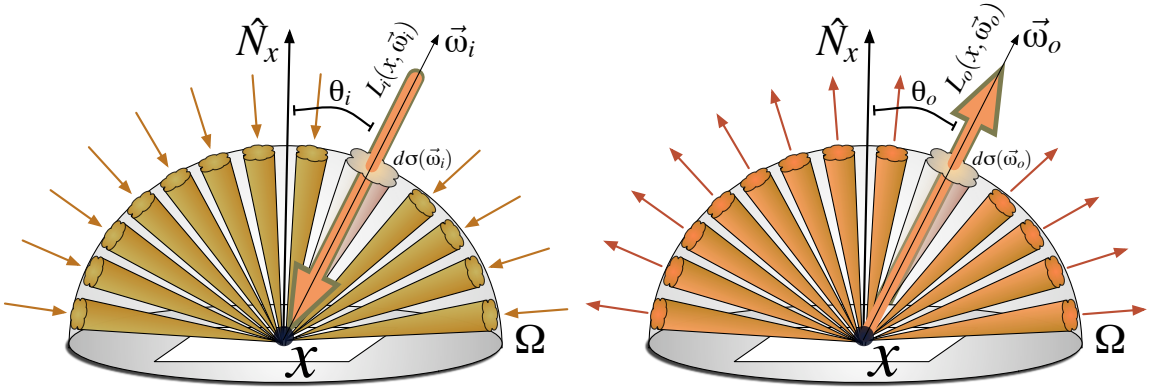


Figure 2.3: Radiance.

the radiance of small solid angles. This can be useful when simplifying equations, since both radiosity and irradiance are three dimensional quantities, only dependent on position.



(a) Irradiance from radiance.

(b) Radiosity from radiance.

Figure 2.4: Irradiance and radiosity in terms of an integration of radiance.

We now mathematically relate these quantities. Irradiance, represented by the letter E, is the total amount of radiance arriving at a point. We represent incoming radiance with L_i and express that it is arriving at a point x from direction $\vec{\omega}_i$ by using notation

$L_i(x, \vec{\omega_i})$. The relation between E and L_i is

$$E(x) = \int_{\Omega} L_i(x, \vec{\omega_i}) \cos(\theta_i)\, d\sigma(\vec{\omega_i}). \tag{2.1}$$

This equation shows that at point x, the irradiance is equal to the total integrated amount of incoming radiance around a hemisphere Ω. Angle θ_i is the angle between direction vector $\vec{\omega_i}$ and the normal ($\hat{N}_x$) of the surface at point x. The $\cos(\theta_i)$ term is necessary to take into account the projection of the flow of incoming radiance against the surface. Finally, $d\sigma(\omega_i)$ is the solid angle measurement (at direction $\vec{\omega_i}$) for the integration of hemisphere (see Figure 2.4(a)).

In this document we will be using the common assumption that all vectors are always pointing *away* from the surface of the point in question. To indicate the direction of the flow of light, we will use the subscripts i for incoming and o for outgoing. Figure 2.4(a) represents a large number of small solid angles integrated over hemisphere Ω. The thick arrow represents one particular instance of these solid angles so that we can see how all the values used in Equation 2.1 are geometrically related. The direction of the flow of irradiance is indicated by the thick arrow and by the fact that the vector representing the direction of the arrow ($\vec{\omega_i}$) has the subscript i.

Radiosity is usually represented by the letter B and is an integral over the total outgoing radiance leaving a point. Outgoing radiance is represented by notation the $L_o(x, \vec{\omega_o})$ (see Figure 2.4(b)). The equation that relates radiance to radiosity is similar to Equation 2.1:

$$B(x) = \int_{\Omega} L_o(x, \vec{\omega_o}) \cos(\theta_o)\, d\sigma(\vec{\omega_o}). \tag{2.2}$$

We can also relate outgoing flux, represented by Φ_o, to radiosity. It can be written as an integral of radiosity over an area:

$$\Phi_o = \int_A B(x)\, dA_x \tag{2.3}$$

If we assume a uniform radiosity over the whole surface, this last equation can be simplied to

$$\Phi_o = BA. \tag{2.4}$$

Another relevant concept is the *bidirectional reflectance distribution function*, or BRDF. This function returns a reflectance value, that is, it describes the way in which a surface point reflects light. For most light interactions, the BRDF is a complicated function that depends on the position on the surface, angle of incidence of the light and of the reflected outgoing angle. In radiometric terms, a BRDF can relate incoming radiance from direction $\vec{\omega_i}$ to outgoing radiance in direction $\vec{\omega_o}$. The BRDF will be represented by the notation $f(x, \vec{\omega_i} \leftrightarrow \vec{\omega_o})$ which means that BRDF f will return a reflectance value ρ for a given point x for incoming radiance in direction $\vec{\omega_i}$ and outgoing direction $\vec{\omega_o}$. The reflectance value ρ is a number between zero and one, and gives the percentage of incoming light that is reflected outwards. A BRDF equation describing the relation between incoming radiance and outgoing radiance is

$$L_o(x, \vec{\omega_o}) = \int_\Omega f(x, \vec{\omega_i} \leftrightarrow \vec{\omega_o}) L_i(x, \vec{\omega_i}) \cos(\theta_i)\, d\sigma(\vec{\omega_i}). \tag{2.5}$$

In this equation the outgoing radiance $L_o(x, \vec{\omega_o})$ in direction $\vec{\omega_o}$ from point x will be equal to the integration of radiance incoming from all possible directions $\vec{\omega_i}$ and weighted by the value of the BRDF for each of those directions (see Figure 2.5).

2.2.2 Solving the System

Diffuse Surfaces

The radiosity approach works only with diffuse surfaces (also called Lambertian surfaces). From a diffuse material all incoming light is equally reflected in all directions (see Figure 2.6). Diffuse surfaces conveniently have a BRDF that is constant: $f(x, \vec{\omega_i} \leftrightarrow \vec{\omega_o}) = \rho(x)/\pi$, where reflectance $\rho(x)$ is the *albedo*, a number that describes the percentage of reflected light, of the surface at point x and π is a factor to maintain energy conservation. This also means that from a diffuse surface the outgoing radiance does not depend on the outgoing angle: it is the same in all directions.

For the classic radiosity algorithm, the system of the energy equilibrium of light within a scene needs to be solved. In other words, we want to know how much radiance is being emitted at each point of each scene surface based on incoming radiance from other points.

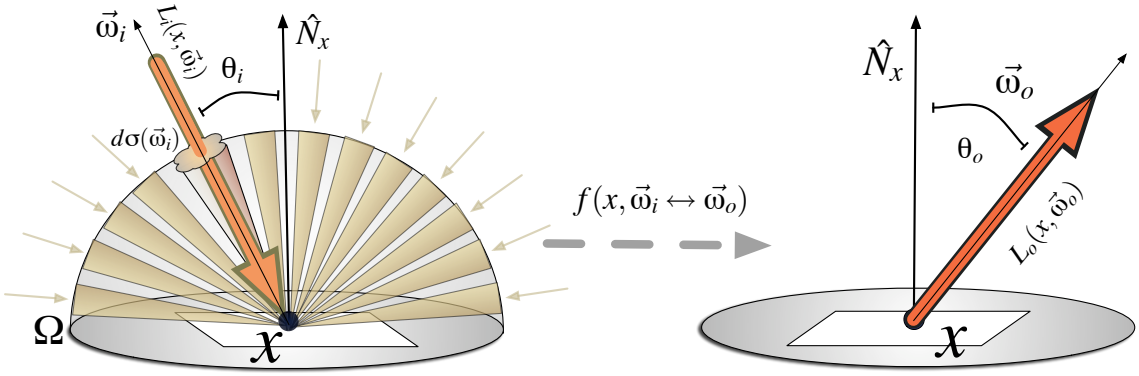


Figure 2.5: Relation between incoming radiance, the BRDF and outgoing radiance.

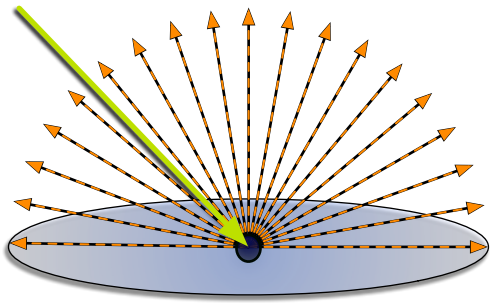


Figure 2.6: Diffuse reflection from a surface.

We can start by applying the diffuse surface simplification to Equation 2.5:

$$L_o(x) = \int_\Omega \frac{\rho(x)}{\pi} L_i(x, \vec{\omega_i}) \cos(\theta_i)\, d\sigma(\vec{\omega_i}). \tag{2.6}$$

Notice that outgoing radiance $L_o(x)$ is now only dependent on point x because the radiance is equal for all angles. We have also substituted the generic BRDF $f(x, \vec{\omega_i} \leftrightarrow \vec{\omega_o})$ with the diffuse BRDF $\rho(x)/\pi$.

We can now redefine the relationships between radiance and radiosity for diffuse surfaces. Equation 2.2 assumes that radiance may vary for different outgoing directions $\vec{\omega_o}$, however, for diffuse surfaces we know that it is constant in all directions. This means that the integration over the hemisphere will be that constant value ($L_o(x)$) multiplied by π due to the cosine term. The relationship is now simplified to

$$B(x) = \pi L_o(x). \tag{2.7}$$

By multiplying Equation 2.6 by π on both sides we can use the relation in Equation 2.7 to create an equation that relates radiosity to incoming radiance. We can also move $\rho(x)$ out of the integral since it is a constant for point x.

$$B(x) = \rho(x) \int_\Omega L_i(x, \vec{\omega_i}) \cos(\theta_i)\, d\sigma(\vec{\omega_i}). \tag{2.8}$$

It is also convenient to change all instances of radiance into radiosity. We can say that the incoming radiance L_i from a given direction is actually the outgoing radiance L_o from another surface. Using this idea and the relation in Equation 2.7 we can substitute the L_i in Equation 2.8 with B'/π to give us

$$B(x) = \frac{\rho(x)}{\pi} \int_\Omega B' \cos(\theta)\, d\sigma(\vec{\omega_i}). \tag{2.9}$$

This equation is the basis for the radiosity map approach that this thesis is presenting. The following subsections finish describing the way in which the classic approach finds the radiosity values for the surface of a scene. The radiosity map approach will be explained in Chaper 3.

The Radiosity Equation

There are two components that make up radiosity B in a given scene: the direct light component and the indirect light component and each can be calculated in different ways. Therefore, it is convenient to separate $B(x)$ into these two components:

$$B(x) = B_e(x) + B_i(x), \tag{2.10}$$

where $B_e(x)$ is the radiosity due to all direct light components and $B_i(x)$ is the radiosity due to all indirect light components. Traditional ray tracing methods can be used to obtain the values for the direct component so the math will not be discussed here. However, the equations for this are shown in Chapter 3.

Calculating the indirect light is not trivial. The classic radiosity approach tries to solve this problem by dividing the scene into patches and changing Equation 2.9 for indirect light into an integration over surfaces that relates each pair of patches instead of an integration over a hemisphere (see Figure 2.7). The equation becomes[2]:

$$B_i(x) = \rho(x) \int_S K(x,y) B(y)\, dA_y. \tag{2.11}$$

This is now an integration over all surfaces in the scene. The radiosity $B(y)$ is given off by a point y on surface S, and $K(x,y)$ is a function that defines the geometric relation between point x and point y, including visibility, distance, and surface orientation. It also includes the π term from the radiance to radiosity substitution. It is defined mathematically as

$$K(x,y) = V(x,y) \frac{\cos(\theta_x)\cos(\theta_y)}{\pi r_{xy}^2}, \tag{2.12}$$

where $V(x,y)$ is equal to 1 if points x and y are mutually visible and equal to 0 if they are not, r is the distance between points x and y, θ_x is the angle between the normal of the surface at point x and the vector that points to y from x, and θ_y is the angle between that same vector and the normal of the surface at point y.

After changing from hemispherical integration to surface integration, Equation 2.10 becomes

$$B(x) = B_e + \rho(x) \int_S K(x,y) B(y)\, dA_y. \tag{2.13}$$

[2]A detailed explanation of the derivation from the hemispheric integration form to the surface integration can be found in [9].

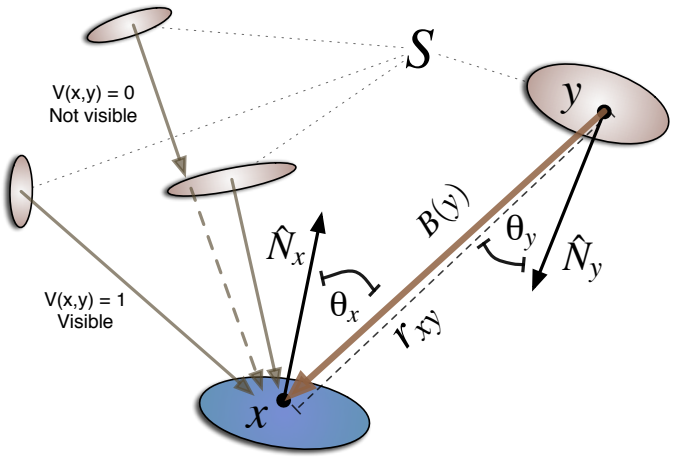


Figure 2.7: Integration of surface areas.

This is called the radiosity integral equation. By using an integral, we are implicitly assuming that the patches are infinitely small. By changing the equation into a sumation instead of an integral, we consider the patches to be small, but not infinitely small. The sumation for each patch i of N total patches with constant radiosity on each is[3]:

$$B_i = B_{ei} + \rho_i \sum_{j=0}^{N} F_{ij} B_j. \tag{2.14}$$

This is the final form of the radiosity equation and it gives a system of equations with N variables, one for each patch. The F_{ij} terms are called patch-to-patch *form factors*. These terms come from the $K(x, y)$ function and must be computed before the system can be solved. They are defined by

$$F_{ij} = \frac{1}{A_i} \int_{S_i} \int_{S_j} K(x, y)\, dA_y\, dA_x. \tag{2.15}$$

Solving the System

To solve the radiosity equation it is necessary to first compute the $N \times N$ form factors and then use them to solve the system of N equations. Solving the system of equations can

[3]A more detailed derivation from the integral form to the sumation form is found in [9].

be done by using the Jacobi method, which is optimized for parallel processing systems or Gauss-Seidel method which is better for sequential processing systems. A brief overview of both methods will now be given.

The Jacobi iterative method solves equations of the type $x = e + Ax$. The idea is to start with an arbitrary point $x^{(0)}$. Then, at each iteration the current point $x^{(k)}$ is transformed into the next point $x^{(k+1)}$ by placing $x^{(k)}$ in the right-hand side of the equation: $x^{(k)} = e + Ax^{(k)}$. If A is a contractive matrix, which means $|Ax| < |x|$ for all x, then the sequence of points $x^{(k)}$ will always converge to the solution of the system (also called the *fixed point*), regardless of the starting position.

The Jacobi method is an optimal method to be run in a system of parallel or distributed processors, since each component of the solution vector $x^{(k)}$ can be computed independently. In the Gauss-Seidel method, each component ($x_1^{(k)}$, $x_2^{(k)}$ ... $x_n^{(k)}$) of the solution vector is processed in sequence. However, once a new component has been generated, it is used in the processing of the next component. Because of this, the Gauss-Seidel method converges twice as fast as the Jacobi method, but at the cost of losing parallelism capabilities.

Both of these methods are excellent for systems such as the radiosity system which is contractive since light reflections are energy conserving. Each iteration models a single bounce of light interreflection in the scene and the equilibrium illumination is the solution to which the iterations converge.

2.2.3 Improvements and Optimizations

Using the classic approach is complicated and time consuming. To get better results, patches need to be as small as possible, meaning that the more patches, the better the results. Unfortunately this can generate large systems of equations and lots of problems when computing the form factors for complex scenes. Furthermore, the number of form factors grows quadratically with the number of patches, and so the naive approach cannot be used for large scenes.

There have been many improvements to the radiosity approach since the original paper was published. One of the first optimizations was the use of the *hemicube* to calculate the form factors [6]. To efficiently compute the double integral Equation 2.15, a hemicube (five

planes forming half of a cube) is constructed around the center of each patch. The faces of the hemicube are divided into pixels and each of this pixels has a precomputed *delta form factor* based on its position on the hemicube. All the visible patches are then projected onto each of the five faces and an approximation to the real form factor of a patch is given by the sum of the delta form factors of the pixels that the patch covers.

In 1988 the Cornell team developed an approach called *progressive refinement* [7]. In the classical approach, solving the equations by using an iterative method is equivalent to gathering illumination contributions in each patch. For the progressive refinement method, the idea is to shoot contributions from one patch onto the other patches, starting with the light sources and going to the patches with the highest energy to shoot. In this way, it is possible to generate a partial solution to the system after only calculating the form factors for one patch. Each shooting step adds more contributions, gradually converging to the complete solution.

Another important improvement to the classical approach was to divide into more patches the areas where more detail was needed, like shadow boundaries or edges. The classical adaptive algorithm was described by Cohen *et al.* and was called *substructuring* [5]. This algorithm would proceed by considering the radiosity variation at the vertices of a patch and would subdivide it if the difference exceeded some threshold. A similar algorithm was *Hierarchical radiosity* by Hanrahan *et al.* [17] in which patches would be subdivided based on their form factors sizes.

2.3 Stochastic Radiosity

In 1986 Kajiya published an analysis of the *rendering equation* [22], a form of the implicit radiance Equation 2.5. This equation described the complete distribution of light in a scene as a transport equation. Describing global illumination in a single equation made it possible to use the concept of stochastic or Monte Carlo integration schemes based on probabilistic methods to calculate additional physical effects. This way, it is possible to calculate illumination due to area lights and produce soft shadows, as well as fuzzy reflections that mimic metals or other reflective surfaces. While the original rendering equation did not include a number of effects, such as participating media (fog or smoke),

this concise mathematical model still covered a range of important effects and led to more principled approaches to light transport.

2.3.1 Monte Carlo Integration

Around the middle of the 20th century, when computers were beginning to be designed, the term Monte Carlo was used to describe mathematical techniques used to simulate stochastic phenomenon by doing statistical sampling. The Monte Carlo approach to solving a problem consists in defining a random variable such that its expected value will be the solution to the problem. Some samples of the variable are generated and the average of these samples is used as an estimate of the true solution.

In the case of computing an integral

$$I = \int_0^1 f(x)\,dx, \tag{2.16}$$

the Monte Carlo approach considers N samples selected randomly over the integral's domain to estimate its value. This is called using an estimator, denoted as $\langle I \rangle$. For Equation 2.16, assuming that the randomly selected variables have a uniform distribution, the estimator would be

$$\langle I \rangle = \frac{1}{N} \sum_{j=1}^{N} f(x_j). \tag{2.17}$$

This estimator is no more than a simple average of values obtained from evaluating the function $f(x)$ for each instance x_i of the random variable x. It can be easily proved that the expected value of this estimator is also the expected value of integral of the function, $E[I] = E[\langle I \rangle]$ [9].

In Equation 2.17 the random variable x has a uniform probability distribution. This means that each instance x_i has an equal probability of being generated. If the random variable had a non-uniform distribution based on some function $p(x)$, then the expected value would be weighed by this function and would now be

$$E[I] = \int_0^1 f(x)p(x)\,dx, \tag{2.18}$$

so it is necessary to divide by function $p(x)$ when defining the estimator. Assuming a non-uniform probability distribution of variables, based on function $p(x)$, the estimator for equation 2.16 is

$$\langle I \rangle = \frac{1}{N} \sum_{j=1}^{N} \frac{f(x_j)}{p(x_j)}. \tag{2.19}$$

Function $p(x)$ is called a *probability distribution function* (PDF). A PDF can be used to increase convergence, that is, reduce the number of samples necessary to give a closer approximation to the real answer. This is called importance sampling and in this case, the PDF should be in some way proportional to the function we want to integrate.

According to Dutré *et al.* [9] there are two main approaches to solving radiosity systems represented by system of linear equations using Monte Carlo Methods: Stochastic relaxation methods and discrete random walks.

2.3.2 Stochastic Relaxation Methods

The basic idea behind stochastic relaxation methods is to solve the radiosity system using an iterative solution method such as Jacobi or Gauss-Seidel while at the same time acquiring the values for the form factors using random sampling [1, 4].

To obtain an approximation to the form factors, rays are shot for each patch and the radiosities obtained are averaged by the number of samples. This essentially gives a summation of radiosities weighed by their probability of being hit, their form factors, and this forms the contractive matrix we can use to find the solution.

2.3.3 Discrete Random Walk Methods

Random walk methods have been used since the 1950s to solve systems similar to those found in the radiosity problem. However, they were not specifically applied to radiosity until 1996 [29]. In a discrete random walk method, there exists a particle that begins in a random state and then is moved through various other states until it is absorbed (see Figure 2.8).

The particle has a normalized probability ϖ_i of initially being in one of the N states and there are different probabilities p_{ij} that the particle might go from one state to another in

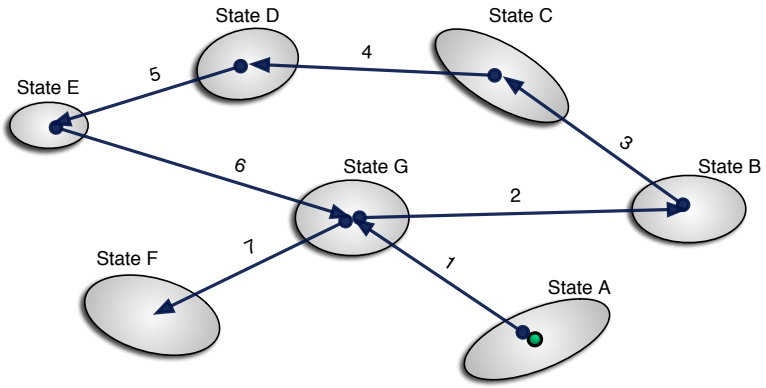


Figure 2.8: Random walk.

particular and there are also probabilities α_i of the particle being absorbed or terminated. The expected number of visits to each state before it is absorbed is called the collision density (as it describes how many times the particle collided with a given state) and it is represented by C_i. The collision density of a discrete random walk process can be found by solving a linear system of equations of the form

$$C_i = N\varpi_i + \sum_{j=1}^{n} p_{ji} C_j. \tag{2.20}$$

We can see that the random walk equation looks similar to the radiosity equation (2.14). We can assume that the particles are photons being shot from the light sources and that each state is a patch in the scene. In a random walk all the initial probabilities must sum to one. The initial probability used can be the emitted light of each patch. To normalize them we can divide the power of each emitted light by the total emitted light power. Then by simulating a number of random walks, the collision density of the walk will give us the light power at each patch.

2.4 Graphics Hardware

Graphics processors originally started as coprocessors with a fixed set of instructions that received a stream of 3D geometry and output a 2D image by rasterization. There has been

a tremendous boom in the industry of video games, visual effects and other related areas and the need for high performance computation in these areas. Thanks to this, GPUs have rapidly grown in complexity and processing power and have become more programmable, while staying relatively inexpensive and widely available.

Current GPUs have two processing stages: The vertex shader and the fragment shader. The vertex shader receives the sequence of vertices and can apply geometric transformations, while the fragment shader receives rasterized information and computes specific shading algorithms. Originally both stages had fixed hardware but first the vertex shader and more recently the fragment shader, where most of the computational power is found, were infused with programmability.

2.4.1 Shading Languages

To facilitate the programming of 3D graphics and GPUs, specialized high level *application programming interfaces* (APIs) have been developed, that give programmers easy to use functions. For 3D graphics, the two most common APIs are OpenGL, created by SGI, and Direct3D, from Microsoft. For GPU programming, the APIs are called shader languages. Two popular ones are Cg from NVIDIA and HLSL, from Microsoft. These shader languages allow the programmer to write simple high level code that will be translated to the GPU's machine code and will run whatever complex shading processes are desired.

In addition, there have recently been APIs that are oriented to running stream processes. These are optimized for running sequences of instructions (called kernels) that are applied on a large set of sequential data. This can be done in highly parallel ways and can be run on clusters of computers, processors or a GPU. Two examples of these languages are Buck's *Brook* from Stanford [2] and McCool's *Sh* from the University of Waterloo [23]. Because Sh combines the concept of stream programming with a shader language, it was the best tool to implement the ideas proposed by this thesis and thus it was selected as the main API.

2.4.2 General Purpose GPU Programming

GPUs have become powerful, but accessible, parallel processors and programming them for general purpose computation (as opposed to just graphics) has become increasingly popular. However, programming for a GPU is not the same as prograaming for a CPU. GPU programs avoid things like conditionals and loops because these were only recently included as part of the hardware and tend to be inefficient. Many algorithms originally created to be run on a CPU have been modified to run on a GPU, where they give better performance. Among these are cryptography, database operations, fast Fourier transformations, the lattice Boltzmann method for fluid flow, segmentation, sound effects processing and neural networks [35].

2.4.3 Ray Tracing on Graphics Hardware

One algorithm that was originally meant for CPUs, due to its large use of loops and conditionals, is the ray tracing algorithm. In 2003, Purcell published a paper on running a GPU ray tracer [28] and since then, there have been a few published implementations of interactive ray tracers, though these ran only on small scenes. Even though at the time GPUs were only meant for non-branching computation, Purcell predicted that GPUs would soon incorporate conditionals and looping and proposed a framework that would use this to efficiently calculate the intersections of traced rays with a scene. This framework involved using a uniform grid as an acceleration structure and keeping all the relevant information in precomputed floating point arrays, accessed as textures. This idea has since been enhanced to use other accelerators [10]. As will be discussed later, the framework we have implemented to calculate radiosity uses a ray tracer based on Purcell's paper.

2.4.4 Radiosity on the Graphics Hardware

GPUs have already been used to calculate or to accelerate the calculation of radiosity in a few published papers. Carr and Hart [4] used precomputed form factors and solved the radiosity matrix on a GPU using Jacobi iterations. *Irradiance caching*, introduced by Ward in 1988 [33], takes advantage of the fact that irradiance usually varies quite smoothly, therefore it can be cached in a coarse data structure. When a lookup is done with this

method, the data can be interpolated for nearby surfaces. This method has been modified to work on a GPU by Gautron *et al.* [12]. Finaly, Coombe emphet al. recently implemented a version of the progressive refinement radiosity algorithm with adaptive sampling on the GPU [8].

2.4.5 Texture Atlases

Textures are arrays (usually of two dimensions but they can be of one or three dimensions) of values that usually contain colors (stored as red, green, blue and sometimes a transparency value) used to shade a 3D object. Textures were initially only used to simulate surface detail that was too fine to be defined by actual geometry. However, as GPUs evolved, more tricks could be played with textures and this became the basis for many shading algorithms. Textures are now oftened used to store intermediate values that will later on be used for a final rendering. In addition, the values stored in textures are not necessarily colors - they can be any arbitrary value.

Usually, each vertex in 3D geometry has a mapping to a rectangular shaped texture map, one for each object, and this mapping is often precomputed or calculated by hand. For certain algorithms it became necessary to have a way to take arbitrary geometry and efficiently map it to a 2D texture. This process is called texture atlasing or texture parameterization (see Figure 2.9). Unfortunately, texture atlasing will usually distort the distribution and shape of the geometry in some way. Consequently, depending on the application, different methods concentrate on preserving certain aspects of the geometry like proportions, shape or connectivity.

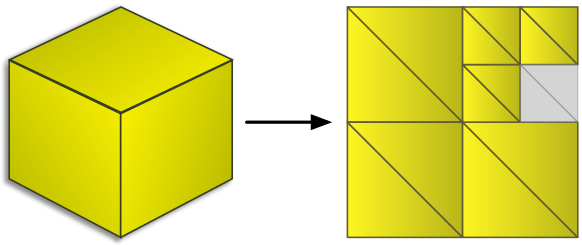


Figure 2.9: Texture atlas.

Texture atlasing has a number of applications. A parameterized 3D object's geometry

can be recalculated in 2D to either reduce or increase its complexity [16]. Texture atlases are also used in 3D painting programs to keep track of the surfaces changes of the objects. Carr and Hart presented an overview of various methods to create automatic parameterizations for texture atlases [3] and proposed a scheme that improved upon these.

Various radiosity methods have used textures to store intermediate parameters or final representations of illumination values. The papers mentioned in Subsection 2.4.4 use textures to store some intermediate parameters. Neilsen *et al.* [25] use a texture atlas to store indices to form factor arrays and then use these to generate the actual surface textures that contain the illumination values. Similarly, the method proposed by this thesis uses textures to store all the intermediate values and the final illumination values for the radiosity solution so that they directly accessible to render the scene. In fact, our entire solution runs on the GPU and generates the texture maps there, without any need to read data back to the host.

No particular existing texture atlasing scheme was used for the implementation presented by this thesis. However, all the papers cited in this section were analyzed to obtain ideas for the implementation of our own scheme. The scheme that was implemented focused mainly on simplicity and ease of coding but lacks certain features. The exact implementation details are found in Section 4.2 while the possible improvements are explained in Chapter refch:conclusion. It is important to mention that texture atlases are not the only possible methods that could be used to parameterize the surface and store the computed radiosity values. Other approaches like poly-cube maps [31] or geometry images [16] could be used. These possible options are also described in Chapter 6.

Chapter 3

The Algorithm

The objective of this thesis is to present an algorithm that can solve the global illumination problem for diffuse interreflection efficiently, avoiding the need to explicitly compute many form factors or solve systems of equations. We also want an algorithm that can be easily implemented and run on an advanced GPU. We assume, however, that we already have a GPU ray tracer algorithm that runs reasonably fast.

Because we want to avoid setting up and solving a system of equations, we will instead use the idea of a linear transport operator. This concept allows us to begin by calculating radiosity due to direct illumination and then incrementally add the contribution of each bounce of indirect light off of the surfaces in the scene.

To be able to compute the illumination, we first distribute sample points over all the surfaces in the scene, similar to the way in which classic radiosity needs to divide a scene into patches. This is done automatically by mapping the 3D geometry into a 2D texture atlas. By using a texture parameterization, all the operations are executed on a simple 2D array. A simple texture atlasing scheme is explained in Chapter 4.

Once the surfaces in the scene have been discretized into points, we calculate the radiosity due to direct illumination for each point and then iteratively compute the component due to additional bounces of light, eventually converging to the complete solution. The following sections explain the concept of the linear transport operator and the mathematics necessary to compute radiosity due to direct and indirect components of the illumination.

3.1 Linear Transport Operator

It is possible to describe the interaction of light with its environment by using linear transport operators. The integral Equation 2.9 takes one bounce of radiosity and transports it into another bounce. The right side of this equation can be considered a transport operator, which we can denote as T. This transport operator takes a distribution of radiance, multiplies it by a reflectance value and then scatters it, generating a new distribution. A function for the distribution of radiosity can be defined as

$$TB = \frac{\rho(x)}{\pi} \int_{\Omega} B \cos(\theta)\, d\sigma_{\vec{\omega}}. \tag{3.1}$$

This equation can be interpreted as saying that radiosity B modified by the transport operator T is equal to the integrated radiosity values one bounce away from surfaces around a point (see Figure 3.1).

Operator T is contractive due to energy conservation. This can be seen mathematically in that the reflected radiosity $TB(x)$ will always be less than the integration of radiosities $B(y)$ over the hemisphere because this integration is multiplied by $\rho(x)$ which is always between zero and one.

We now define the total scene radiosity recursively in terms of the transport operator:

$$B = B_e + TB. \tag{3.2}$$

This recursive equation can be expanded into a series because the transport operator T is contractive. If we operate on both sides of the equation by T, we get a new value of TB:

$$TB = TB_e + TTB. \tag{3.3}$$

We can substitute this to the original equation over and over:

$$B = B_e + TB_e + TTB,$$

$$B = B_e + TB_e + TTB_e + TTTB,$$

$$B = B_e + TB_e + T^2B_e + T^3B_e + \ldots + T^nB.$$

This is called the *Newman series* and is an important concept that path tracing and random walk approaches use [9, 22, 32]:

$$B = \sum_{n=0}^{\infty} T^n B_e. \tag{3.4}$$

Geometrically, this expansion represents light being emitted from one surface, then being distributed to other surfaces over and over again (see figure 3.2). We can approximate a solution by truncating this series to a finite number of terms. Since T is contractive, the omitted terms will decrease to zero as the number of bounces included in the summation increases.

The more terms this series is expanded to, the closer it converges to the real solution. If we can apply this operator directly we do not need to solve a system of equations.

3.2 Acquiring Lighting Contributions

To account for one execution of the transport operator we need to acquire lighting contributions from all surfaces that surround each point in a scene. Calculating these contributions can be done using Equation 2.9. In classic radiosity, the hemispherical integration in this equation needs to be changed to a surface integration to obtain an equation that is dependent on all the other surfaces. However, since this requires each pair of patches to be related, patch to patch form factors need to be computed. For this algorithm we will assume that each patch is represented by a single point and integrate over a hemisphere around that point. As we are integrating over a hemisphere and not relating patches, form factors do not need to be explicitly computed.

To obtain the exact value of the incoming radiance at each interval in this integration we must shoot a ray, find the first surface point it hits and obtain the value of radiosity at that point. Shooting this ray is represented by function $y = r(x, \vec{\omega})$ and, because the contribution will be from the first surface hit, there is no need to account for a visibility term. For a given solid angle σ with direction $\vec{\omega}$ we can shoot a ray in this direction and obtain a point y. We can also separate the computation of radiosity B into first computing the irradiance E and then multiplying by the reflectance ρ/π. In addition, we should move

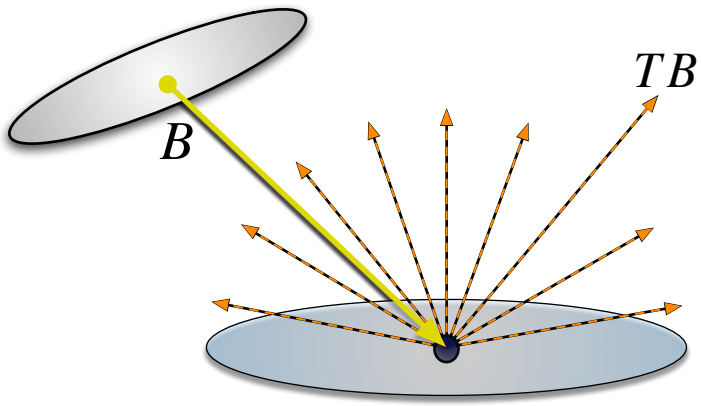


Figure 3.1: The transport operator.

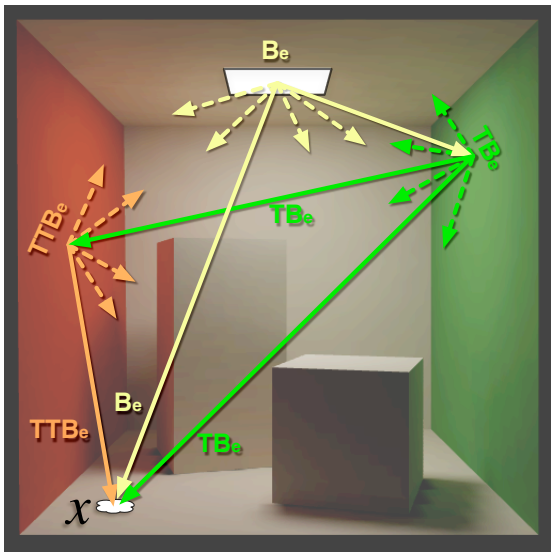


Figure 3.2: Rays of light distributed through the transport operator.

the division by π to inside the integral since this will be an important factor when using the Monte Carlo method. The irradiance function for the contribution of indirect light is

$$E(x) = \int_{\Omega} B(y) \frac{\cos(\theta_x)}{\pi} \, d\sigma(\vec{\omega}). \tag{3.5}$$

And the radiosity for indirect lighting contributions will be

$$B(x) = \rho(x) E(x). \tag{3.6}$$

This equation can be estimated by a summation of random ray samples (see Figure 3.3(b)). To obtain faster convergence, it is convenient to use an importance PDF to generate the random rays. This means that instead of generating completely random rays, the probabilities are weighed towards where the contribution of incoming radiance affects the integrand more. In this case, light approaching the surface from the direction of the normal has a greater contribution than light coming at an angle, due to the effect of the cosine term. A cosine-weighed probability distribution function, that is, one that generates more rays pointing towards the normal than towards gracing angles, gives better convergence. The cosine-weighed probability distribution function is defined as $p(x) = \cos(\theta_x)/\pi$, and it cancels out the equivalent terms in equation 3.5. The final estimator is therefore simply

$$E(x) = \mathbf{E} \left[\frac{1}{M} \sum_{i=1}^{M} B(y_i) \right]. \tag{3.7}$$

In this equation, operator **E** means the *expected value*. The more samples that are taken, that is, the higher M is, the closer the result of the estimator will be to its expected value. This form for Equation 3.5 helps to explain why the explicit computation of form factor terms is not required. A form factor is a percentage that relates how much radiosity flows from one patch to another. With Monte Carlo sampling, this percentage is accounted for by the probability that each ray has of hitting a given patch.

3.3 Initial Lighting Contribution

If we sample randomly around the hemisphere of each point assuming that every patch that is not light emitting is originally in a state of total darkness ($B = 0$) we will need a

large number of samples before converging to a presentable approximation. If we assume that light emitting patches (i.e. light sources) will have small surfaces in relation to the rest of the scene, it is better if we calculate the initial transport of light by executing a point to patch integration from each point to the light source. As in the classic approach, we can divide radiosity at a point into two components, that of the initial lighting contribution (B_e) and that due to subsequent bounces of light in the scene ($B_\imath$):

$$B = B_e + B_\imath. \tag{3.8}$$

The term $B_\imath$ can be calculated using the procedure described in the previous section. For B_e we can use equations 2.11 and 2.12 to calculate the integration of all surfaces that emit light:

$$B_e(x) = \frac{\rho(x)}{\pi} \int_{S_\ell} B_\ell(y) V(x,y) \frac{\cos(\theta_x)\cos(\theta_y)}{r_{xy}^2}\, dA_y. \tag{3.9}$$

The emitted light component of the radiosity at point x is equal to the integration of the self emitted radiosity $B_\ell(y)$ over the surface area of all light sources S_l multiplied by the surface reflectance $\rho(x)/\pi$ at point x. As this is a point to surface integration, form factors have to be taken into account: the inverse r_{xy} squared law for distances and both the projected area of the reflecting surface ($\cos(\theta_x)$), and the projected area of the light source ($\cos(\theta_y)$ onto the light vector (see Figure 3.4(a)). The function $V(x,y)$ gives the visibility between point x, on the surface, and point y, on the light source.

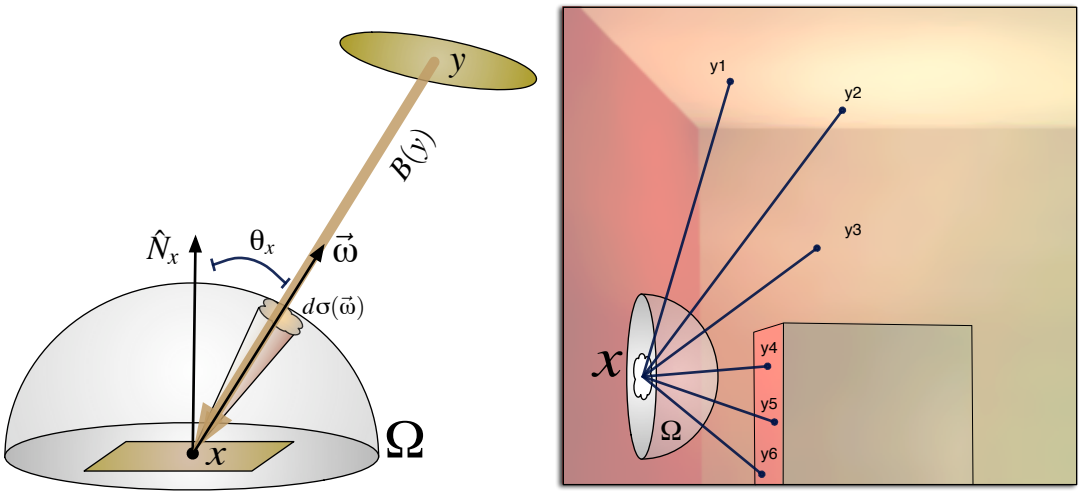
As in the previous section, it is convenient to separate Equation 3.9 into two steps. First the integration of direct light B_ℓ, which gives us an irradiance value E_e:

$$E_e(x) = \int_{S_\ell} B_\ell(y) V(x,y) \frac{\cos(\theta_x)\cos(\theta_y)}{r_{xy}^2}\, dA_y. \tag{3.10}$$

Then we multiply E_e by the reflectance ρ/π to obtain radiosity B_e:

$$B_e(x) = \frac{\rho(x)}{\pi} E_e(x). \tag{3.11}$$

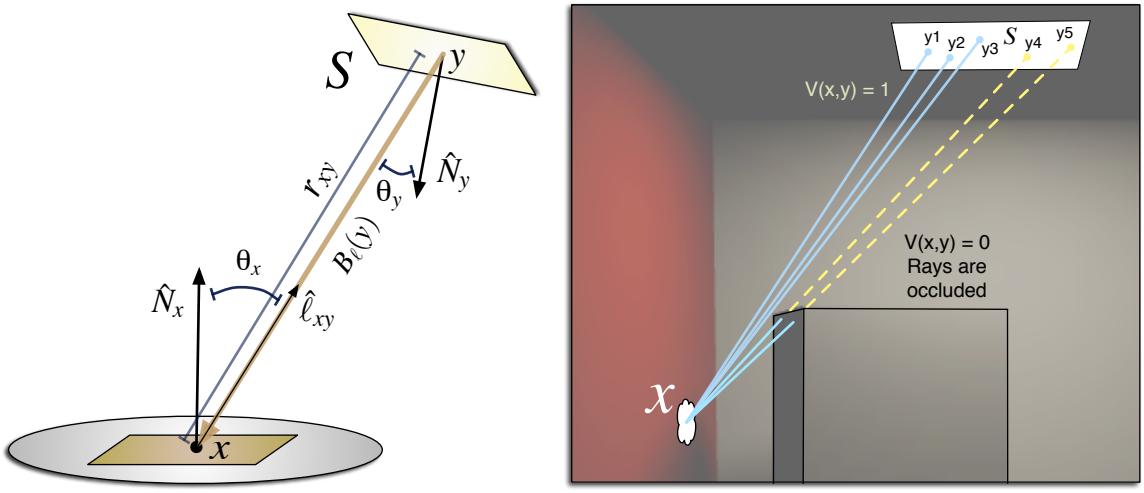
Again, we can use a Monte Carlo approach to approximate the irradiance function $E_e(x)$. We define an estimator whose expected value will be the contribution of light from the direct light source. This estimator is based on Equation 3.10 but now point y is drawn



(a) Integration over the hemisphere.

(b) Shooting cosine distributed rays.

Figure 3.3: Calculating indirect illumination.



(a) Integration of light surface.

(b) Shooting rays to the light source.

Figure 3.4: Calculating direct illumination.

from a uniform probability distribution function $p(y)$. We use a uniform sampling of the light source area, which gives us $p(y) = 1/A$ [9]. We can also assume that the light source has equal radiosity accross its surface and use the relation shown in Equation 2.4 to have flux instead of radiosity: $B_\ell(y) = \Phi_\ell/A$. Inserting the PDF $p(y) = 1/A$ in the denominator to account for the probability distribution and substituting B_ℓ with Φ_ℓ/A cancels out the A and leaves the equation in terms of flux Φ_ℓ.

$$E_e(x) = \mathbf{E}\left[\frac{1}{N}\sum_{i=1}^{N}\Phi_l V(x, y_i)\frac{(\hat{N}_x \cdot \hat{\ell}_{xy_i})(\hat{N}_{y_i} \cdot -\hat{\ell}_{xy_i})}{\pi r_{xy_i}^2}\right]. \tag{3.12}$$

In this equation $\hat{N}_x$ is the normal of the surface at point x, $\hat{N}_{y_i}$ is the normal of the light source at point y_i and $\hat{\ell}_{xy_i}$ is the vector that points from x to y_i. All three are unit vectors. The cosine terms $\cos(\theta_x)$ and $\cos(\theta_y)$ from Equation 3.10 are now represented by dot products $(\hat{N}_x \cdot \hat{\ell}_{xy_i})$ and $(\hat{N}_{y_i} \cdot -\hat{\ell}_{xy_i})$, respectively, since this is the notation used in the pseudocode given in Chapter 4 and the actual implementation. To know the value of visibility function $V(x, y)$, a ray in the direction of ℓ_{xy} needs to be shot from point x using a ray tracer to find if there are any occluders (see Figure 3.4(b)). The combination of these terms is a point to patch form factor that relates point x with the area of the surface of the light source A.

3.4 Iterating the Series

In Section 3.2 we explained that an indirect light contribution can be obtained using a Monte Carlo method, assuming a cosine distribution PDF with equation Equation 3.7. If we multiply the right side of this equation by the albedo ρ of the point we are analysing, we get radiosity instead of irradiance and the equation becomes a Monte Carlo approximation of the transport operator defined in Equation 3.1:

$$TB \approx \rho\frac{1}{M}\sum_{m=1}^{M} B. \tag{3.13}$$

This converges to a contribution of light, and a summation of these terms converges to the real light equilibrium solution according to the Neumann series shown in Equation 3.4.

We can use this approximation to calculate the contribution from each bounce of light. However, this means that for K bounces we need to iterate $M \times K$ times. To reduce the number of iterations, we present a different approximation to the Neumann series that only iterates M, but that generates a small error for contributions of light after the second bounce. Instead of each transport operation being approximated by an average of values, each transport operation will obtain only *one* value:

$$T'B = \rho B. \tag{3.14}$$

Instead of obtaining the average of contributions for each transported value, we will account for $K = M$ bounces of light and average all the transported contributions:

$$B \approx \frac{\sum_{k=1}^{K} T'^{k} B}{K}. \tag{3.15}$$

For $\lim K \to \infty$ this series converges to the same value as the Neumann series. However, for numbers smaller than infinity there are small differences. If we assume that the radiosity due to the first bounce of light is given by B_e and expand the Neumann series for K terms we get

$$B = B_e + TB_e + T^2B_e + T^3B_e + \ldots + T^KB_e. \tag{3.16}$$

Expanding the series used by this framework, in terms of the original operator T, gives

$$B = B_e + \kappa_1^{(K)}TB_e + \kappa_2^{(K)}T^2B_e + \kappa_3^{(K)}T^3B_e + \ldots + \kappa_K^{(K)}T^KB_e, \tag{3.17}$$

where $\kappa_k^{(m)}$ is a number between zero and one that reduces the value of each contribution k due to having the averaging done over the whole series. The value for κ_1 is always equal to 1 so the first term in the series is equal to that of the Neumann series. The equation that defines the value of $\kappa_k^{(m)}$ for each iteration for terms $k > 1$ is recursive:

$$\kappa_k^{(m)} = \frac{\sum_{n=k}^{m} \kappa_{k-1}^{(n-1)}}{m}. \tag{3.18}$$

In this equation m is the total number of iterations. The equation states that the κ value for each term k at each iteration m depends on the sum of the κ values for the previous term for all previous iterations. The error $\epsilon_{\kappa_k^{(m)}} = 1 - \kappa_k^{(m)}$ for each term is reduced as M

increases, but converges towards zero slower as the k value increases, that is, the higher order terms converge slower.

Using the series given by Equation 3.15 for a limited but large number of iterations gives us a solution that is accurate for the contributions due to the first few bounces but loses power as more bounces are added. However, since the operator T is contractive, each additional bounce contributes less and less, so the error to the total solution becomes smaller for each additional bounce. This gives us a good approximation for only M iterations.

We also account for the radiosity (B_e) due to the first bounce of light requiring iterating before a close approximation can be obtained. To have an interactive environment we need to reduce the number of iterations for B_e since we need to have this value before we can render an initial approximation. The solution is to first calculate B_e based on a small number of iterations and then continue to iterate B_e as the transported terms in the series shown in Equation 3.15 are also iterated. This will generate another small error term but this error will also converge towards zero as more iterations are executed.

3.5 Programming steps

The sequence of steps necessary to calculate a radiosity map for a scene with the algorithm presented by this thesis are as follows:

1. Create a texture atlas of the scene. This automatically divides the surface of the scene into discrete sample points and map these points to a 2D array.

2. *Initialize* values for direct illumination, indirect illumination and total radiosity at these discrete points.

3. Create an initial radiosity contribution B_e based on irradiance from direct light. For each sample point:

   (a) Calculate its direct lighting contribution by shooting N_{init} rays into random locations on the light source.

   (b) Compute the average of the radiosity contributed by these rays and store these results into the direct illumination array.

4. *Iterate* radiosity contributions from additional bounces ($B_\imath$) and continue iterating radiosity from direct light (B_e). For each sample point:

   (a) While fewer than M samples have been generated, calculate one indirect lighting sample by shooting a random ray over the hemisphere of that point and adding to an array of accumulated indirect contributions the total radiosity (the sum of the B_e and $B_\imath$ components) generated by the surface point that was hit.

   (b) Compute the average of the accumulated indirect radiosity contributions $B_\imath$ and store this result in the indirect illumination array.

   (c) While fewer than N samples have been generated, calculate direct lighting contribution by shooting a ray into a random location on the light source and accumulate this value.

   (d) Compute the average of the radiosity contributed by these rays and store these results into the direct illumination array.

   (e) Add both indirect and direct lighting components and store the result in the total radiosity array.

Chapter 4

Implementation Details

The radiosity algorithm presented in the previous chapter is the main contribution of this thesis, however, an important part of the contribution is also the way in which various programming elements have been combined to form an efficient framework to render radiosity as a series of stream programs for GPUs. This chapter presents the structure of that framework and has detailed pseudocode and descriptions explaining the more relevant parts. Figure 4.1 shows a general view of the framework.

A file containing a scene description and the corresponding geometric data for the objects within that scene are read as inputs. This data goes into a *scene* module which bundles it up into data structures that the other modules can easily access. The data is first passed to the *texture atlas* module where the whole geometry is mapped into a 2D representation of its surfaces, a texture map. This map is sent back to the scene module, then the geometry for the scene is sent to the *ray tracer* module where it is placed into an *accelerator structure*. In the case of the implementation done for this thesis, this structure is a uniform 3D grid [28], but other structures can be used, as long as they are implemented for a GPU. After these modules have been initialized with the data and the necessary structures have been precomputed, the *user interface* (UI) module begins a rendering loop. In this loop the *radiosity* module is first called to initialize various texture maps that will contain the illumination information. The scene is then rendered to the screen using an initial approximation of the global illumination. Only an initial approximation is used to allow the program to have interactive frame rates for the user

to move around the scene. While the interface is idle, the radiosity module will iterate to converge the solution towards the real value and the UI module will smoothly merge the initial approximation to the real value. As long as the light source and the geometry in the scene remain static, the user may navigate around the scene in real time since the radiosity solution is view-independent and rendering new view points only requires access to the radiosity textures. Once the light source or the geometry are modified, it is necessary to recompute the radiosity values and update the lookup textures.

The following sections give more details of this framework. In the first section there is a discussion on the selection of the main tool used for this implementation, the Sh language, and why it is adequate. Also in that section is an explanation of the way GPUs are programmed. Section 4.2 describes the texture atlas module. Then, Section 4.3 discusses the ray tracer and scene modules. Finally, the radiosity mapping implementation is explained as well as its interaction with the user interface in Section 4.4.

4.1 Sh as a Tool for GPU Radiosity Maps

The radiosity maps framework was written in C++, using OpenGL as its 3D API and GLUT for a simple user interface.The shader library selected for this implementation is the Sh Metaprogramming Language [23]. The name "Sh" comes from the word *shader* (not to be confused with "shell") because the original application for Sh was to write high level GPU shader code. Sh is a C++ library based on templates. It has a steep learning curve, but the resulting code is usually quite elegant. One of the main advantages of Sh is that it can be used as a shader language *and* as a stream programming language because it has multiple backends. In other words, the code it generates can be executed on a GPU, a CPU or even on a cluster of PCs (though some modifications are necessary). Sh also has various useful graphics utilities like a complete linear algebra library and an object file loader for the *obj* format.

The main use for Sh in this theses is as a shader API. A shader program simply refers to a series of instructions that will be executed in one of the two programmable pipeline stages of the GPU: the vertex shader or the fragment shader. The vertex shader is usually in charge of receiving geometry information, applying transformations to it and generating

a 2D array of potential pixels (fragments) for the fragment shader to receive. The fragment shader will then apply shading computations or any other complex calculations per pixel and output a color value. As explained in the background chapter, these programmable stages can also be used to compute other things more complex than simple geometric transformations and shading calculations including, for example, finding the intersection of a ray with a set of triangles or calculating radiosity and direct lighting values.

The fragment shader is usually used to do the heavy calculations. However, to avoid generating shader programs that are too complex or that require the use of too many conditionals or loops, the programs are subdivided and are executed as passes. The output of the fragment shader is usually stored in a pixel buffer that is then output to a window on the computer monitor. To store this output in a texture instead of a window is not completely trivial and the output path of the GPU must be configured adequately. A popular open source library named *RenderTexture* by Harris [18] was used for this operation.

4.2 The Scene and Texture Atlas Modules

The scene module is in charge of handling every aspect of the geometry in the scene. It also includes a parser to read text files that have the scene description. From these files the scene module receives the geometry information for all the objects in the scene, including any applied translations, rotations and scaling transformations. It then stores this information in large textures that function as arrays of vertices, normals and other attributes. In current graphics hardware, no more than four values can be retrieved from each texture cell, however, since most of the quantities stored in textures have three dimensions, this is not a problem. In addition to specific object details, the scene module also reads in other general characteristics such as area light sources, background color, field of view and initial viewing position and direction.

From the scene module, the vertex data is sent to the texture atlas module where it is mapped to a 2D array (see Figure 4.3(a)). There has been a fair amount of research in the field of creating texture atlases and a good source of information is Carr and Hart's paper on *Meshed Atlases for Real-Time Procedural Solid Texturing* [3]. In this paper Carr and Hart talk about various methods for distributing triangles from a 3D description to a 2D

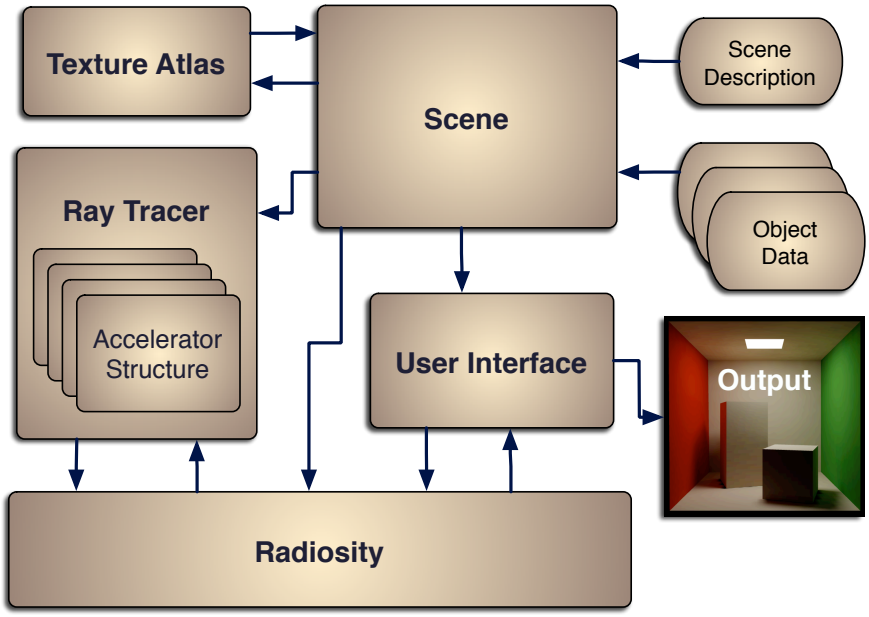


Figure 4.1: General framework.

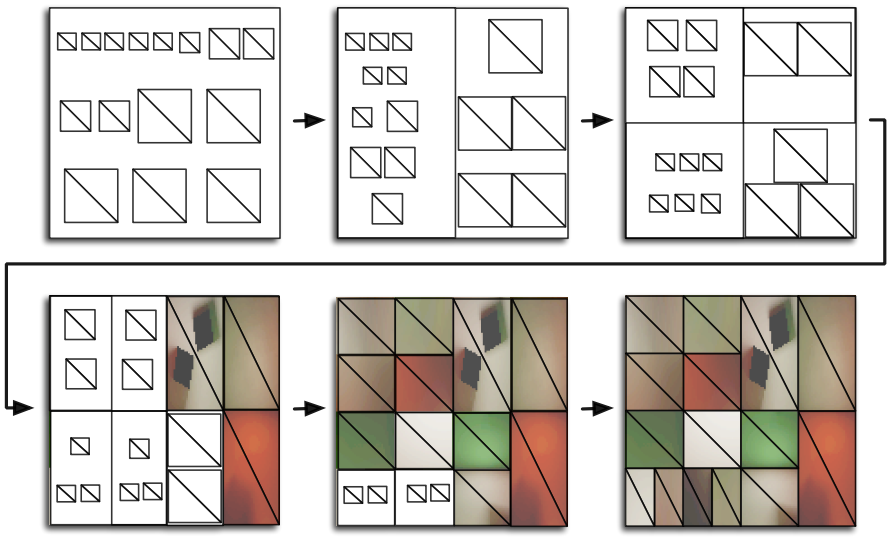


Figure 4.2: Texture atlas generation.

array. This paper was used as a basis to the implementation of a simpler texture atlasing scheme whose goal was only to demonstrate that global illumination implemented this way would work. For elaborate applications, a more robust texture atlasing scheme should be implemented.

In the scheme used for this framework the triangles are packed into pairs of neighboring triangles and the pairs are ordered by area (see Appendix C.1.1 for pesudocode).The groups of triangle pairs are then recursively divided within the texture space using a simple greedy algorithm that takes into account area size and triangle numbers, trying to evenly distribute the areas of the triangles between the two sides, but without letting too many triangles gather on one side. The texture space where they will be mapped onto is also divided first vertically, then horizontally until only one triangle pair is in each division (see Appendix C.1.2 for pseudocode).

This scheme is easy to implement and was done this way due to time constraints. The result is an atlas that takes advantage of the whole texture space and tries to keep the proportions of triangles close to that of the ones in the original geometry. However, no attempt to avoid triangle deformation was made. Fortunately this is not a problem, except for the case of long, skinny triangles.

When accessing the textures for the final rendering the texels are bi-linearly interpolated to obtain a smooth look. Because unconnected triangles are next to each other on the texture atlas, this would normally cause some of the colors from the nearby triangles to bleed together. Carr and Hart proposed reducing the texture lookup by half a texel, which works fine for pairs of triangles. This scheme, however, leaves some triangles on their own. These "loner" triangles need a special lookup table that indicates what triangle it should be connected to on the diagonal edge (or if it has no connection) so that the proper interpolation can be done.

4.3 The Ray Tracer

The ray tracer is an important part of the radiosity algorithm. Having a ray tracer that runs on the GPU allows the whole process to be run within the GPU since it is by shooting rays that samples of the radiosity around the hemisphere of each point will be computed.

The ray tracer for this implementation is based on the one published by Purcell [28]. For the ray tracer to work efficiently a good accelerator structure must be built, in this case it is a uniform 3D grid. Normally it is convenient to have a grid of dimension of the cube root of the number of triangles, however this data structure will eventually be represented as a 2D texture array and so it is better to use dimension values that are powers of two, since 2D textures sometimes have to abide to this.

After defining the dimensions of the uniform grid, the module proceeds to make a bounding box around each of the triangles and to see in which of the voxels in the grid the triangle is in. Then, for each voxel, the program calls a triangle-box intersection routine to make sure that the triangle is really inside the voxel. Purcell's approach to store this data involves having three textures. One is the 3D texture that represents the grid and is referenced by each voxel's coordinates. Each cell in this texture contains pointers to a second texture. The second texture contains lists of pointers to triangle data. There is one list of pointers for each voxel and each list is delimited by a special symbol (-1 in this implementation). These pointers are indexes that reference another texture, one that contains the specific vertices and normals for each triangle.

Shooting a ray requires an input of a position and a direction. With this information the module can define a position on the grid from where it will start traversing it, if it was hit. Traversing the grid involves two loops. The first loop checks each voxel that is in the path of the ray. For each voxel, the module looks up the 3D grid texture and obtains the pointer to the second texture. The second loop goes through each triangle pointer found in the second texture until it reaches the delimiter symbol. With each pointer, the module can access the triangle information it points to and make a ray-triangle intersection test to see if that triangle was hit. The output of the ray tracer is the ray length ℓ, a barycentric coordinates pair (uv) that describe exactly where the triangle was hit, and a triangle identification number (-1 when nothing is hit). These values are stored in a 4-tuple variable called *luvi* and are returned as output.

The ray tracer can also work with movable groups of objects. For this, a different grid is created for each group of objects. Each group may be set to contain only one object to create the ability of moving individual objects. For each grid, one pass is done and the output *luvi* values are stored. When the next grid is processed, a check is made to see if

the new ray length value ℓ is shorter than the previous one. If it is, then the new *luvi* value is used, as this means that an object that was closer was hit in this grid. When a grid is transformed by a movement (rotation or translation) these transformations are stored in a matrix. Instead of recreating the grid, each ray shot throughout that grid is transformed by the inverse of the given transformations, which has the same effect as transforming the geometry inside that grid. Obviously, a transformation matrix needs to be stored for each grid.

The pseudocode for various shaders is listed in the sections below, however, the shader that executes the ray tracing passes (referred to as *RT*) will not be listed. This is because it goes beyond the scope and contribution of this thesis. For more specific implementation details consult [27] and [24].

4.4 The Radiosity Module and User Interface

The user interface (UI) module is in charge of joining all the modules and allowing the user to interact with the scene. The first task of the UI module is to call the scene module to initialize the geometric data. Once all the initialization has been executed, the UI module goes into a rendering loop where it will be calling the radiosity module as needed to calculate the radiosity maps. It will also be receiving input from the user to modify the viewpoint, move objects or other possible actions. The UI module interacts with the radiosity module by executing three computing stages: *initialization*, *iteration* and *final rendering*. Each of these stages runs a series of shader passes that together perform the radiosity calculations. In the following sections there is a detailed description of each stage and an explanation of the types of passes used in each one.

4.4.1 Shader Passes

The radiosity module uses three types of passes, defined by the types of vertex shaders (see Figure 4.3): *Geometry to texture* (G2T), *texture to texture* (T2T) and *geometry to 2D image* (G2I).

The geometry to texture pass receives geometry, texture coordinates and other attributes obtained from the scene module and generates a texture map based on the infor-

mation selected in the fragment shader. This vertex shader is listed in Appendix C.2.1.

The texture to texture pass reads in one or more textures, processes them, and generates another texture of the same size with new information (see Figure 4.3(b)). The vertex shader pseudocode for this pass is listed in Appendix C.2.2. It is extremely simple as it only acts as a pass-through sequence for the fragment shader. This kind of vertex shader is used for general purpose computing executed on a GPU. It involves sending into the vertex shader a single triangle of geometry using an orthographic projection (that is, with no perspective) that is aligned such that only the center part of the triangle will be visible in the viewport. The texture coordinates are also conveniently placed so that the center of the triangle has exactly the range $(0, 0)$ to $(1, 1)$. This means that the fragment shader will receive a sequence of texture coordinates that will be used to index the textures used as data arrays (see Figure 4.4).

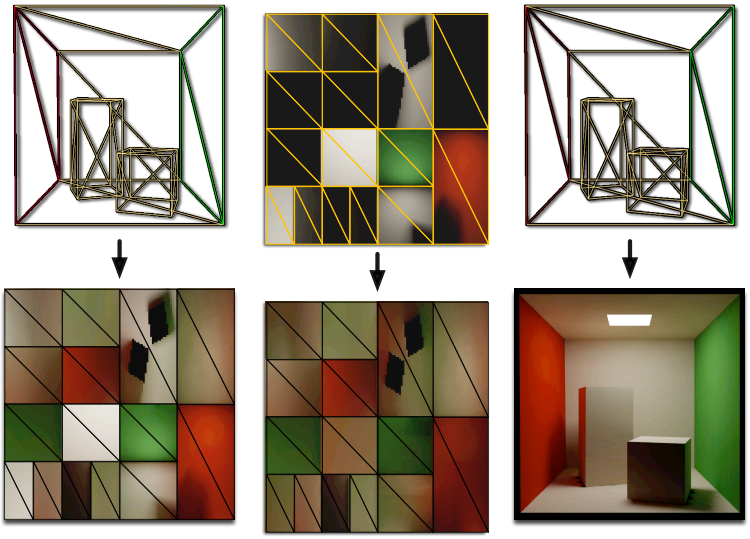
The geometry to 2D image pass receives geometry, texture coordinates and attributes and additionally may read one or more texture maps. It then uses all these to generate a 2D image representing the 3D scene (see Figure 4.3(b)). This vertex shader is described in Appendix C.2.3.

4.4.2 Radiosity Calculation Stages

This section describes in detail each of the stages where radiosity is calculated. As explained previously, each stage executes a number of shader passes. For those passes that use either the G2T or G2I passes, the fragment shader is relatively trivial and an explanation shall be omitted. However, for those passes that use the T2T pass, the fragment shader is in charge of doing all the processing so the shaders will be explained below.

Initialization

The initialization step is called the first time the program is run or immediately after any geometry, including light sources, in the scene is moved. The first step in the initialization process generates arrays (textures) that contain surface positions, colors and normals according to the texture atlas using G2T passes. There are also accumulator arrays that store the sum of all the direct ($Eu_e(x)$) and indirect ($Eu_i(x)$) irradiance components before



(a) Geometry to Texture (G2T).

(b) Texture to Texture (T2T).

(c) Geometry to 2D Image (G2I).

Figure 4.3: Vertex shader types.

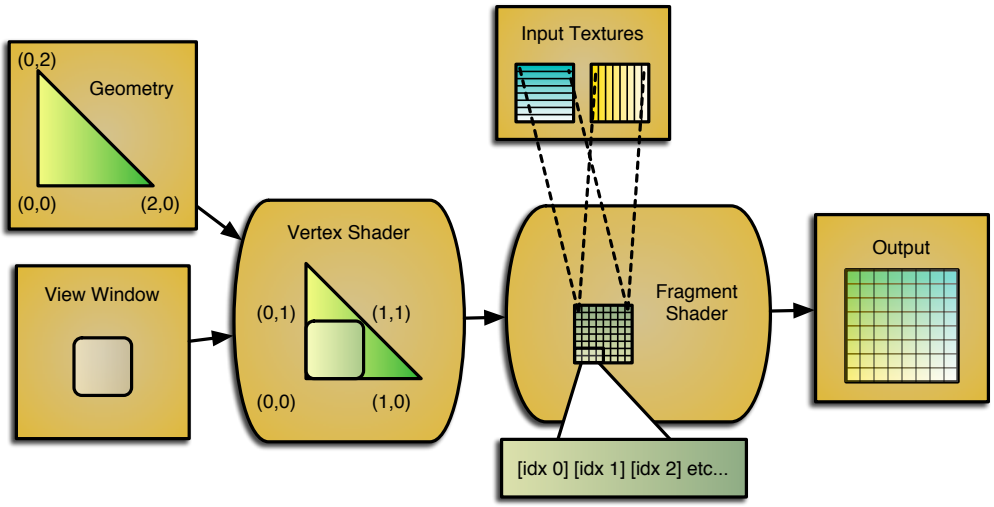


Figure 4.4: Using the vertex shader to pass texture coordinates as an array of indices to textures.

they are divided by the number of rays that have been shot, and these must be cleared, as well as the counters used to know how many rays have been shot. The clearing pass (CL) is a $T2T$ pass with a fragment shader that simply outputs the value $(0, 0, 0)$ to all of the output texture. It is important to note that all the texture arrays used to store color values should use full floating point textures, otherwise there will be a noticeable banding of colors due to the lack of precision.

Once all the values have been cleared, some initial approximations need to be calculated. If the hardware was fast enough, it would be possible to calculate the complete radiosity solution during this step and just skip to the final rendering step without looping through the iteration step. Unfortunately, current hardware is not fast enough for this implementation and so it is convenient to simply generate a fast initial approximation and progressively calculate the complete solution while the user interface is idle. This initial approximation is the radiosity due to direct illumination B_e.

To generate each sample, three $T2T$ passes are needed. The first pass, referred to as GLR (pseudocode in Appendix C.3.1) will generate a light ray randomly, that is, a ray going from point x towards a random point in the light surface[1]. The second pass will be the ray tracer pass (RT) and will return whether an object was found on the way to the light surface or not. The final pass, named DL (pseudocode in Appendix C.3.2), will add the contribution of the direct light, taking into account the mathematics discussed in Section 3.3.

Once all the samples have been generated, another pass can average them and multiply the result by the reflectance ρ and divide by π to get the correct initial approximation of the direct light contribution to radiosity B_e. The indirect light contribution to radiosity B_i will start at zero, sot it must be cleared. Given this, the total radiosity $B(x)$ can be initialized with only the direct light contribution. See Pseudocode 4.1 for the complete listing of steps for the initialization stage.

During the initialization stage it is be possible to execute additional steps, which will accelerate the process of convergence. If a number of rays are precomputed for each texel,

[1] Although multiple light sources would be treated similarly, the implementation presented is specifically for a single rectangular flat surface. It was made this way to avoid geometrical complications, however the same math applies to having multiple sources or to having different shapes, only the generation of the samples becomes more complicated.

Algorithm 4.1 Initialization stage.

$x \leftarrow G2T$ // `Generate positions map.`
$N_x \leftarrow G2T$ // `Generate normals map.`
$\rho(x) \leftarrow G2T$ // `Generate colors map.`
$Bu_e(x) \leftarrow CL$ // `Clear direct radiosity accumulator component.`
$Bu_\iota(x) \leftarrow CL$ // `Clear indirect radiosity accumulator component.`
for $i = 1$ to N_{init} **do**
  $\ell_{xy} \leftarrow GLR(x)$ // `Generate array of random rays to the light surface.`
  $V(x, y) \leftarrow RT(\ell_{xy})$ // `Shoot ray to find visibility and distance.`
  $Eu_e(x) \leftarrow Eu_e(x) + DL(l_{xy}, V(x, y), N_x)$ // `Accumulate light sample.`
end for
$B_e(x) \leftarrow \frac{\rho(x)}{\pi} \frac{Eu_e(x)}{N_{init}}$ //`Average and store direct lighting approximation.`
$B_\iota(x) \leftarrow CL$ //`Clear indirect radiosity contribution`
$N \leftarrow N_{init}$ //`Update direct ray counter.`
$M \leftarrow 0$ //`Initialize indirect ray counter.`
$B(x) \leftarrow B_e(x)$ //`Initialize total radiosity.`

these rays can be used for extremely quick passes and thus an initial value of radiosity from indirect lighting can be calculated. In practice, using an NVIDIA GeForce 6800 we were able to store the necessary random rays for 75 passes of indirect illumination calculation and had those passes executed at the initialization stage. The passes are the same as those in the iteration stage so they will be explained there. For Chapter 5 where the results are presented, the convergence timings were measured using precomputed rays, giving better results than otherwise.

Iteration

The iteration step will be called as many times as necessary to converge to a good radiosity approximation. The values N and M used in the pseudocode listings are iteration counters for the direct and indirect lighting components respectively. Each execution of this stage will run a sampling pass for either components as long as their iteration counters are below their predefined limits. In general, the direct lighting component converges faster than the indirect component and so requires fewer iterations. The exact iteration limits used for this implementation are presented in Chapter 5.

Instead of having arbitrary numbers that limit the amounts of iterations, we could test for convergence by computing the standard error and comparing agains a arbitrarily defined error value. To do this we would have to compute the standard deviation at each pixel and divide by the square root of the current amount of iterations. However, this would require gathering the values of all the texels in the radiosity array and this is a relatively slow operation, therefore it was decided that for this framework only predefined limits would be used.

As the direct lighting calculation was already described in the initialization pass we will now explain the indirect lighting calculation sequence. First a random ray must be generated. This ray should point from the surface position x to a place over the hemisphere surrounding the surface where point x is. A cosine distribution function is used to generate this ray, as explained in Section 3.2. The formulas to do this are derived in [30] and [26]. Given two uniform random variables ε_0 and ε_1 between zero and one, a ray l will have a cosine distribution if its x, y and z components are defined as:

$$l_x = \cos(2\pi\varepsilon_0\sqrt{\varepsilon_1}),$$

$$l_y = \sin(2\pi\varepsilon_0\sqrt{\varepsilon_1}),$$
$$l_z = \sqrt{1-\varepsilon_1}.$$

Once this ray has been generated with the random cosine distribution ray generator, referred to as GCR (pseudocode in Appendix C.4.1), the RT pass is used to see what the ray hits. With the information from the ray tracing pass, an indirect lighting calculating pass, referred to as IL (pseudocode in Appendix C.4.2), is executed to add the contribution of the surface point that was hit. The IL shader is extremely simple and only needs to access the radiosity emitted by the point (y) hit by the randomly shot ray. If nothing is hit, then the background color is taken as the radiosity contribution.

Once both radiosity components have been iterated, the new approximations of both contributions are added and stored in an array containing the total radiosity for each sample point. See Pseudocode 4.2 for the complete listing of steps for the iteration stage.

Algorithm 4.2 Iteration stage.

$l_{xy} \leftarrow GLR(x)$ // `Generate array of random rays to the light surface.`
$V(x,y) \leftarrow RT(l_{xy})$ // `Shoot ray to find visibility and distance.`
$Eu_e(x) \leftarrow Eu_e(x) + DL(l_{xy}, V(x,y), N_x)$ // `Accumulate new direct light sample.`
$N \leftarrow N + 1$ //`increase direct light ray counter.`
$B_e(x) \leftarrow \frac{\rho(x)}{\pi}\frac{Eu_e(x)}{N}$ //`Average direct lighting approximation.`
$l_{xy} \leftarrow GLR(x)$ // `Generate array of cosine distributed random rays.`
$y \leftarrow RT(l_{xy})$ //`Shoot ray to find the first surface to be hit.`
$Eu_i(x) \leftarrow Eu_i(x) + IL(y)$ // `Accumulate new indirect light sample.`
$M \leftarrow M + 1$ //`increase indirect light ray counter.`
$B_i(x) \leftarrow \rho(x)\frac{Eu_i(x)}{M}$ //`Average indirect lighting approximation.`
$B(x) \leftarrow B_e(x) + B_i(x)$ //`Add and store both contributions.`

In addition to accumulating the radiosity values, a blending process between iterations has been added. Instead of using the radiance array $B(x)$ for the final output, a blended radiosity array is used. This array is initially cleared at the same time as the position, color and normals are generated. From then on, it will always be gradually fading into whatever value $B(x)$ has from its previous value. This way, although in the background

the real radiosity of the current scene is being computed, the output uses the previous calculation of radiosity and slowly fades into the new one as it is calculated.

Final Render

The previous two subsections dealt with generating the textures that are mapped to the surfaces of the objects. Once the blended radiosity texture is initialized, the geometry to 2D image shader pass is used to render the final output image which maps the $B(x)$ texture to the geometry. Since the $B(x)$ texture is being progressively improved while the program is idle, the user is free to navigate around the scene even though the final result may not have been reached yet. The scene will be displayed with the current approximation and this approximation will improve over time.

To properly display the values generated by the radiosity mapping calculations, the colors must be tone mapped. This means modifying the output colors (c) so that they are in the correct range for the computer monitor to display. There are various alternatives to doing this but the one selected for this framework is

$$c = \frac{c}{k + c}.$$

A constant k is used to manually tune the tone mapping. The values for k found to give visually pleasant results were between 0.1 and 0.02.

Chapter 5

Results

The results generated from this framework will now be presented. The first section of this chapter gives a series of comparisons of a scene well known in the computer graphics literature and uses these comparisons to validate the results of this method. A small analysis of the results from these comparisons is presented at the end of that section. The second section gives some rendering times using specific hardware and shows additional scenes that demonstrate the potential of the algorithm.

5.1 Validation

The framework we present is intended to generate a 3D image that can be interactively manipulated. Although the focus was placed on having a fast algorithm rather than a predictive one, the mathematics that this framework uses are based on real world physics and therefore the results are expected to be realistic.

To validate how close a 3D image of a scene is to its real world counterpart it is necessary to build the real scene and take careful measurements of the illumination values reflected from it. Then the values can be compared and the validity of the 3D image can be decided.

The people at Cornell University have carried out extensive research regarding this type of validation [15]. They built the well known *Cornell box*, which is a simple scene consisting of a small box and a long box inside a larger box that is missing the front side. The larger box has the left side red, the right side green. The rest of the surfaces, including the smaller

boxes, are gray. A small square surface light is shone into the top of the box to illuminate it. The information gathered from the measurement data of this box is publicly available. Unfortunately, most of this data is based on high precision spectral measurements that go beyond the scope of this framework. More importantly, most of the data they have is for a version of the Cornell box that has a perfectly specular surface, a mirror, as the material for the long box.

The image shown in Figure 5.1(a) is sample RGB image of Cornell University's baseline renderings and we found it to be the most adequate to make our comparisons. We used the geometric data publicly available from Cornell University to set up scene parameters that could be used by the radiosity maps framework to display the Cornell box. As we were unable to adequately transform the spectral data given as input, we manually tried to find the closest approximation of lighting and reflectance values for the surfaces. The image rendered by our approach is shown in Figure 5.1(b). This image was generated by running our framework using a texture map with a resolution of 512×512, 200 direct lighting samples and 2000 indirect lighting samples. These numbers are higher than those shown for the sample scenes shown in the next section since we wanted to evaluate the correctness of the algorithm in this stage and therefore needed the most accurate result possible.

Another way to validate the radiosity map method is to use another a different rendering algorithm. If the two algorithms are different enough, then obtaining identical results from both is a good indication that they are both correct. This is because the probabilities of them both having an error that generates the same output is low due to the methods being so different. With this in mind, we also entered the inputs used for our approach to a previously built path tracer and generated Figure 5.1(c) by letting the path tracer run for a day, shooting 3000 paths per pixel and with each path reflecting a maximum of 5 times.

5.1.1 Histogram Comparison

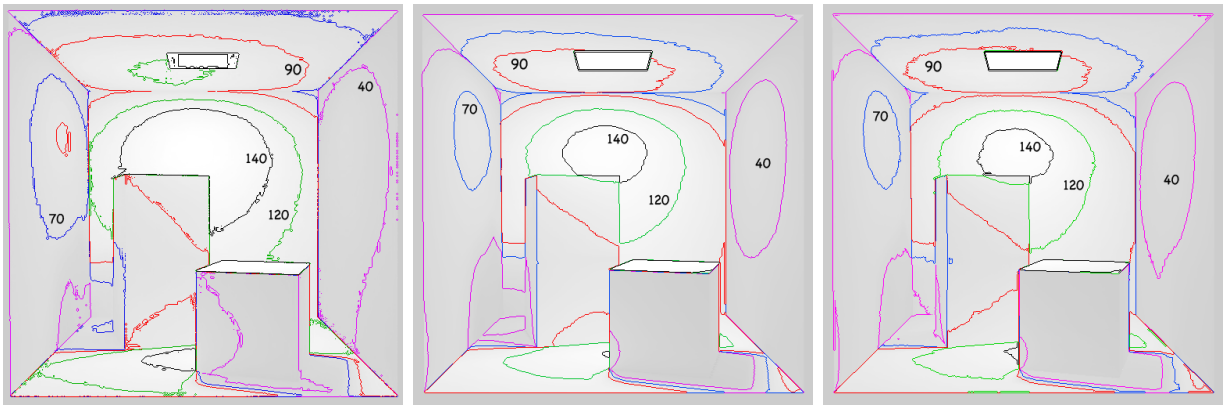
Using an image editing program we generated the histograms of the luminosities of these three images. A luminosity histogram shows a graph of the count of each pixel with the same perceived brightness. The left side of the graph represents the darker pixels while the right side represents the lighter ones. In Figure 5.3 we can see that the three histograms

(a) Original Cornell box.

(b) Radiosity mapped Cornell box.

(c) Path traced Cornell box.

Figure 5.1: A comparison of the Cornell box images.



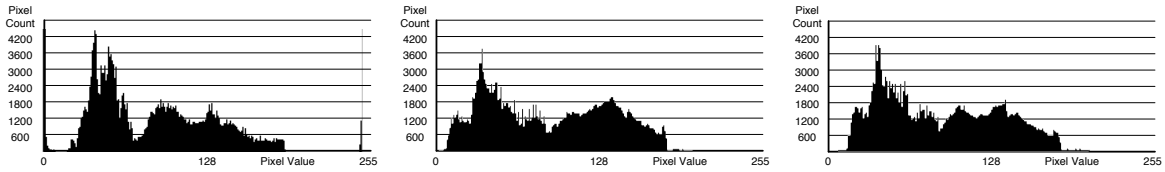
(a) Original Cornell Box contours.

(b) Radiosity map Cornell box contours.

(c) Path traced Cornell box contours.

Figure 5.2: Contours for the Cornell box image at brightness values of 40, 70, 90, 120 and 140.

follow the same pattern of peaks and valleys. This means that, while the exact brightness may vary, the general illumination of the scenes is the same.



(a) Original Cornell Box histogram.

(b) Path traced Cornell box histogram.

(c) Radiosity map Cornell box histogram.

Figure 5.3: Histograms for the Cornell box images.

5.1.2 Contour Comparison

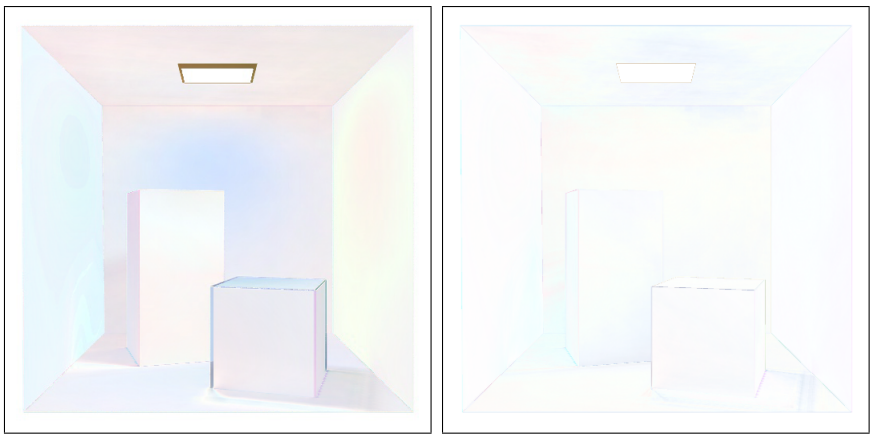
We can trace contours at a given pixel brightness for each of the images. A contour line is drawn wherever the brightness of the image goes through a given threshold. In Figure 5.2 we can see the contours for pixels passing at boundaries of 40, 70, 90, 120 and 140 (on a scale of 0 to 255). It is clear to see that the three images have similar contours, but the brightness values are shifted.

5.1.3 Pixel Differences

Another measure of comparison is to show the pixel differences between the two images. Figure 5.4(a) shows the differences between the radiosity map rendered box and the original Cornell box, while Figure 5.4(b) shows the differences between the radiosity map box and the path traced version. The darker areas represent greater differences of colors while the lighter areas have similar or identical values[1].

In the comparison with the original Cornell box, most of the image has at least a small difference in color. We can see that the wall on the right and the front face of the tall box

[1] A difference image commonly has dark areas for no difference and light areas for large differences, however, the colors have been inverted for these images to make them printer friendly.



(a) Pixel differences between radiosity map image and original Cornell image.

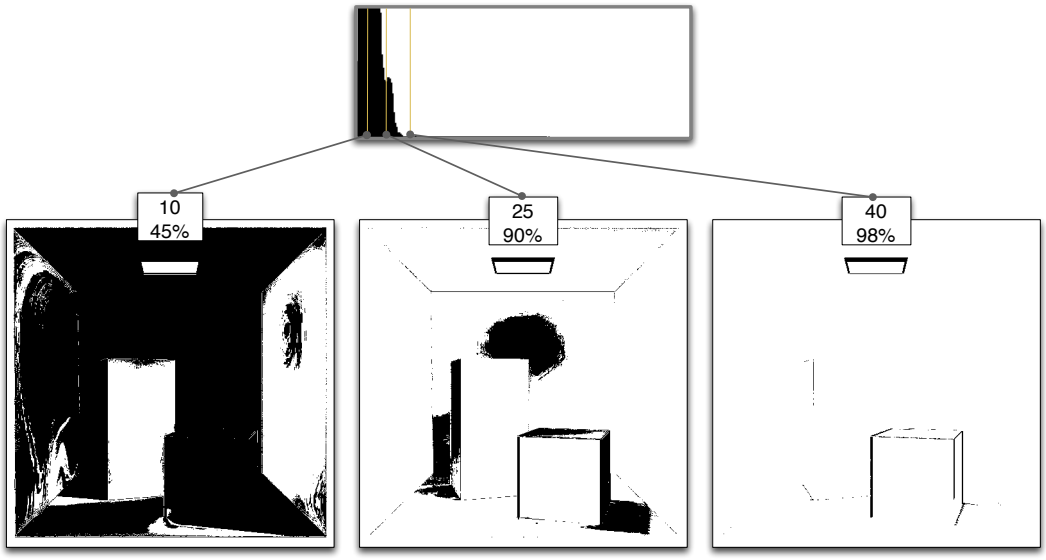
(b) Pixel differences between radiosity map image and path traced image.

Figure 5.4: Pixel differences between radiosity map generated image and two other rendered images.

have similar colors. However, in the comparison with the path traced image the pixels are almost identical.

To better observe the magnitude of the pixel differences, we can plot the histograms for the differences and also render threshold images for specific difference levels. The images shown in Figure 5.5 show pixel difference thresholds for levels of 10, 25 and 40. The scale used for the levels is 0 for no difference and 255 for total difference (black against white or vice-versa). In these images black pixels represent those pixels that have a higher pixel difference than the level shown. In other words, the blacker the image is, the more pixels have a higher difference than the selected level.

At a level of 40 of pixel difference, both images only show black pixels in object edges, due to misalignment of the image renderings, and where resolution artifacts exist, mainly, the area where the right face of the small box meets the floor. Almost all the pixel differences (98% and 99.8% for the original box and the path traced box respectively) are below this threshold level.



(a) Original Cornell box

(b) Path traced Cornell box

Figure 5.5: Pixel difference thresholds.

At a level of 25, the original Cornell box has some noticeable differences, mainly on the shadows of the boxes and the highlight on the back wall. The path traced version still shows no noticeable differences at this threshold. Only 10% of the pixels in the differences with the original Cornell box have this difference level or higher.

Finally, at a level of 10, the differences with the original Cornell box show large areas that have this difference or higher. In fact approximately half of the pixel differences are above this level of threshold. The differences with the path traced version barely start to show at this level.

5.1.4 Validation Analysis

From the luminosity histograms and the countour images, it is clear that there are shifts in the perceived brightness levels. Shifts in brightness were expected between the radiosity map box and the original box. In fact, there is a clear difference in brightness between the image presented by Cornell and the other two images. This is probably due to different tone mappings, though, and is not a reason for concern.

The images of pixel difference tolerances show other discrepancies between the images. The differences in the edges, found at tolerance level 40, are related to sampling resolutions when generating the scene and are expected. The texture resolution artefacts are caused by having a texel in an area that is mostly obscured by an object but that, due to having a low resolution, sticks out under the object.

At threshold level 25 the comparison with the original Cornell box shows a relatively large area of pixel differences. One possible reason is due to discrepancies regarding the light source. The light shape for our implementation is a square of variable size, however, the one used for the original Cornell box is not exactly square in shape. More importantly, the data for the light given are a reflectance value and wavelength values. Our framework accepts a flux value and an RGB color for the light, which is not as physically accurate, but is a necessary compromise for real-time rendering. In addition, the values for tone mapping were tweaked manually, based only on perception. Therefore, it is really only reasonable to compare these scenes qualitatively.

Even though these differences exist, they are very small. The discrepancies that are above a threshold of 25 but below 40 are those that generate our greatest concern, but even

these are only a 10%-15% difference in perceived brightness. It is clear that the differences with the path traced version are minimal, having less than 4% of perceived difference in all pixels.

5.2 Sample Scenes

This section presents some of the results of this framework. For the Cornell box scene, discussed in the previous section, we present a broad number of timings that show how different configurations affect performance. For all other scenes we have one or two timings that give an idea of how the algorithm scales.

All the results shown here were generated using a computer running on Windows XP and equipped with an Intel Pentium 4 processor running at 2.53GHz with 512MB of RAM and an NVIDIA GeForce 6800 graphics card with 256MB of video RAM. The scenes were rendered with 200 iterations for the direct lighting component and 800 for the indirect lighting component. These values were not varied since each iteration always takes exactly the same time for the same scene and configuration and it was found that these numbers gave visually pleasing results.

For each scene the following values are shown: precomputation time, number of triangles, texture resolution used, time to precompute 75 fixed rays and time to converge. The precomputation time includes the time to load scene geometry, time to compute the uniform grid and triangle lists for the ray tracer and time to create the texture atlases for resolutions of 64×64, 128×128, 256×256 and 512×512. All times are in seconds.

Cornell Box

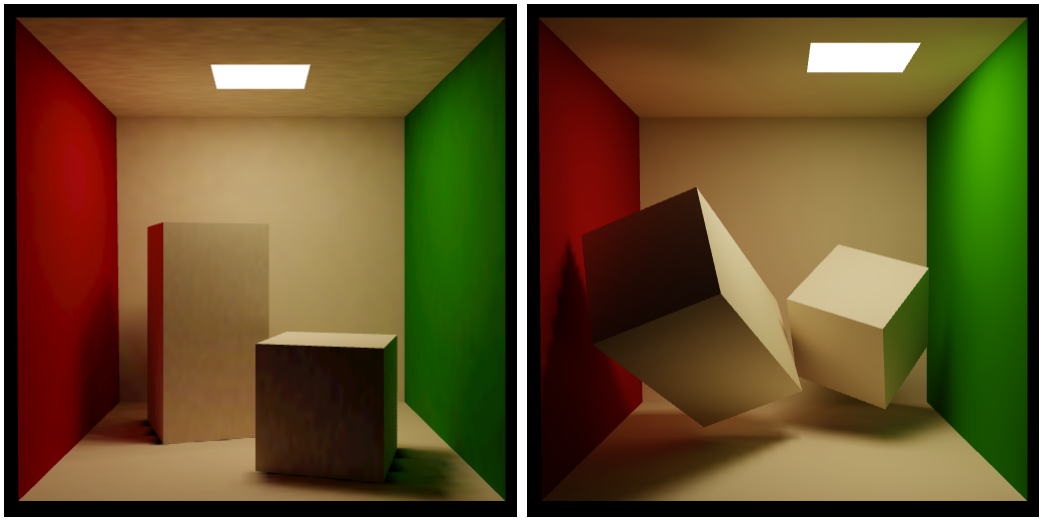
The final rendering of the Cornell box is shown in Figure 5.1(b). Its texture atlas can be seen in Figure 4.3(b). Figure 5.6 shows the progression of the illumination on the scene (using a texture atlas of 128×128 as more iterations are computed and Table 5.1 shows the timings for the converged image using various texture atlas resolutions. It is important to point out that Figure 5.6(a) shows a global illumination approximation that can be used with real interactive rates.

(a) After .263 seconds and 90 iterations.

(b) After 3.733 seconds and 205 iterations.

(c) After 7.611 seconds and 410 iterations.

Figure 5.6: Progression of global illumination.



(a) Different random samplings at each texel.

(b) Moving the two internal boxes and the light source.

Figure 5.7: Alternative renderings.

Number of triangles	30
Precomputation time	2.946s

Texture Resolution	**Solution time**	**Precomputing 75 rays**
64×64	8.276s	0.478s
128×128	15.865s	1.816s
256×256	37.971s	3.431s
512×512	144.815s	9.444s
(random sampling)		
128×128	38.115s	—
(3 moving objects)		
128×128	24.689s	—

Table 5.1: Cornell box scene timings.

In addition to the standard timings, for the Cornell box scene we present the time taken using a more randomized sampling scheme (see Figure 5.7(a). For the timings shown in the previous table, the random directions generated for the indirect lighting component are the same for every texel per pass. Using a totally random direction in which each texel uses different random directions per pass reduces cache coherency and makes the computation process slower.

The ray tracer has a mode that allows the movement of objects without having to recompute the acceleration structures (see Section 4.3). Our Cornell box scene was configured to have three moving objects: the small box, the long box and the large box (see Figure 5.7(b)). Having the ability to move these objects, however, slows the raytracer somewhat, and rays cannot be precomputed. The timing for a 128×128 texture resolution for three moving objects is shown at the bottom of Table 5.1.

Unfortunately we were unable to find performance data for other similar algorithms for the same scene for comparison purposes, except for the path tracer. The path tracer took about a day to complete the image shown in Figure 5.1(c) using 3000 samples per pixel running on a machine equipped with dual Pentium 4 2.3GHz processors. However, this is a relatively unoptimized implementation, and does not use the same GPU ray engine as was used for our radiosity map implementation.

Spheres and Cube

The *spheres and cube* scene shows a red, a yellow sphere and a gray cube floating inside a box with green and gray walls (see Figure 5.8 and Table 5.2). This scene has a background color with a slight greenish hue, which generates a brighter indirect illumination component for those surfaces facing the open end of the surrounding box. This scene was used to test the algorithm on smooth surfaces. The shading is still flat looking due to slight discontinuities in the texture atlas.

Number of triangles	*478*
Precomputation time	*3.278s*

Texture Resolution	**Solution time**	**Precomputing 75 rays**
128×128	39.2436s	3.623
256×256	104.166s	8.799

Table 5.2: Spheres and Cube scene timings.

Toys

The *toys* scene shows a teddy bear with some cubes and a ball (see Figure 5.9 and Table 5.3). With the teddy bear, scene introduces an object with more complex geometry. This means that a texture atlas with a higher resolution is needed to guarantee that every triangle has at least one pixel assigned to it and that the larger triangles have enough resolution to show shadow details.

Number of triangles	*3426*
Precomputation time	*5.854s*

Texture Resolution	**Solution time**	**Precomputing 75 rays**
128×128	86.1171s	7.009s
256×256	229.452s	20.133s

Table 5.3: Toys scene timings.

Drinking Teddy and Jeep

The *drinking teddy* (Figure 5.10 and Table 5.5) and *jeep* (Figure 5.11 and Table 5.4) scenes are two other examples of the radiosity map approach at work. The drinking teddy scene has a prominent bluish hue, generated from a blue background.

Number of triangles	*3998*
Precomputation time	*4.5863s*

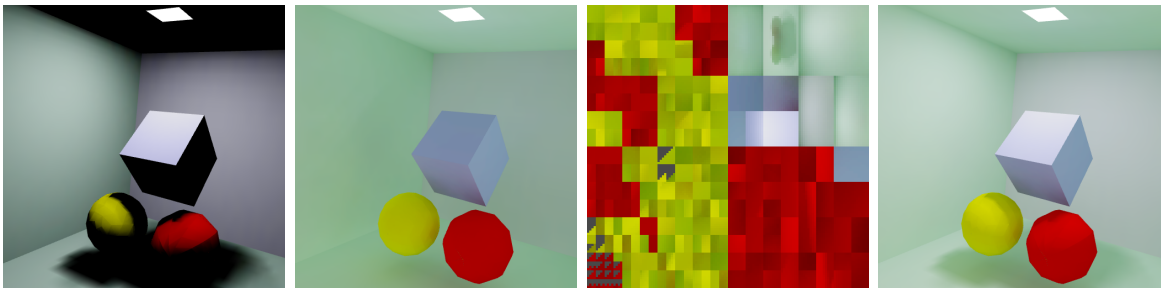
Texture Resolution	**Solution time**	**Precomputing 75 rays**
256×256	260.632s	23.036s

Table 5.4: Jeep scene timings.

Number of triangles	*3534*
Precomputation time	*4.082s*

Texture Resolution	**Solution time**	**Precomputing 75 rays**
256×256	297.17s	26.1535

Table 5.5: Drinking teddy scene timings.



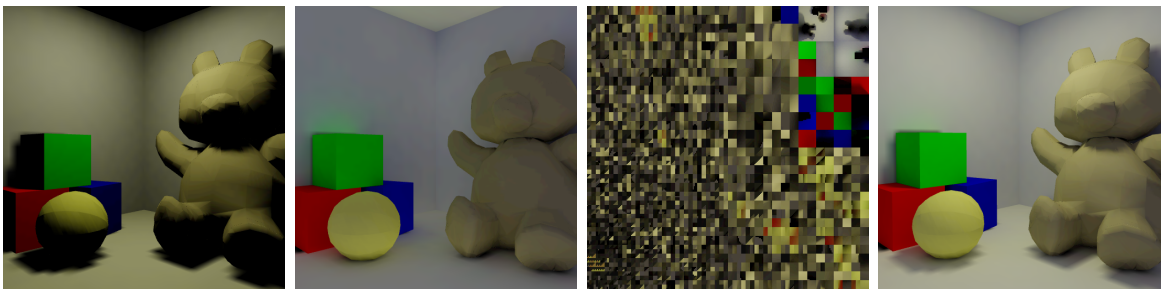
(a) Direct lighting component.

(b) Indirect lighting component.

(c) Texture atlas.

(d) Final output.

Figure 5.8: Spheres and cube.



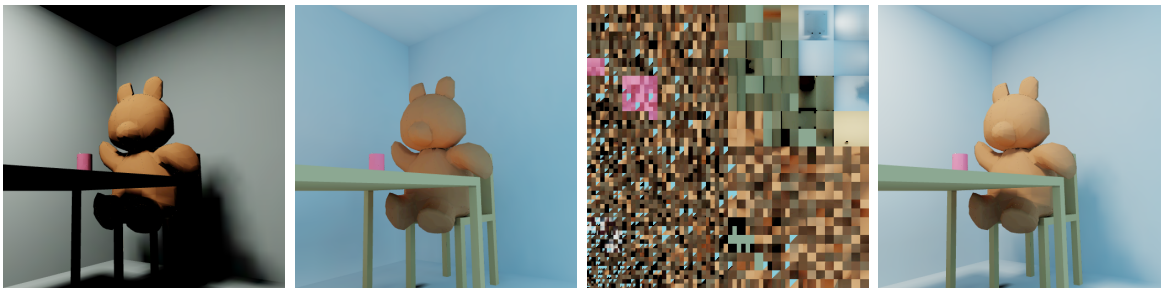
(a) Direct lighting component.

(b) Indirect lighting component.

(c) Texture atlas.

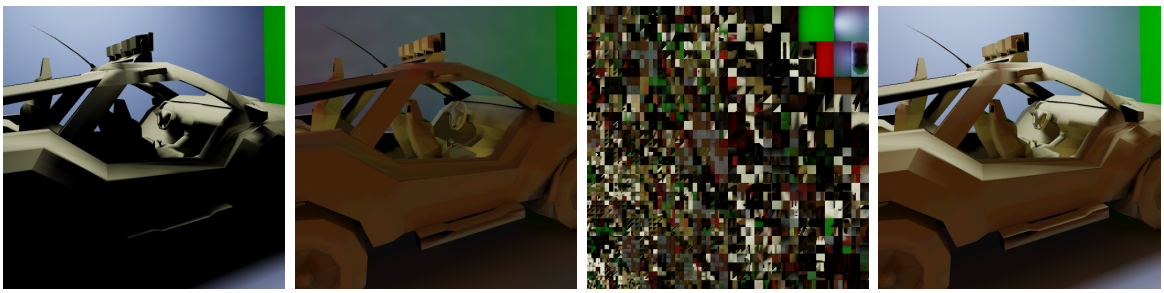
(d) Final output.

Figure 5.9: Toys.



(a) Direct lighting component. (b) Indirect lighting component. (c) Texture atlas. (d) Final output.

Figure 5.10: Drinking teddy.



(a) Direct lighting component. (b) Indirect lighting component. (c) Texture atlas. (d) Final output.

Figure 5.11: Jeep.

Chapter 6

Conclusion

This thesis has presented an algorithm for efficiently calculating the radiosity of a scene that can be executed completely within a GPU architecture by using a GPU based ray tracer and a texture atlasing scheme to store the radiosity values. We believe that this design has a lot of potential but we have also realized that there are some limitations that have to be overcome. In this chapter we present these benefits and limitations as well as some future work that can be accomplished to continue this research.

6.1 Benefits

Having a design that is completely run within a GPU has many advantages. GPUs are advancing in speed and complexity much faster than CPUs are and that means that this framework will be able to accept more complex geometry much sooner. Also, because the resulting radiosity is stored in textures, the information can be used immediately to render the scene, which makes this an excellent algorithm for interactive applications. The fact that a texture atlas is used has an added benefit, it makes the computation time depend more on the resolution of the textures used to compute radiosity and less on the complexity of the scene. This allows for texture based adaptive resolution schemes to improve the quality of the solution.

Although it is necessary to pre-compute acceleration structures for the ray tracer and a texture atlas to map the surfaces in the scene, by using a Monte Carlo approach calculating

and storing form factors is avoided. By creating groups of precomputed data for each object or group of objects, it is possible to change the positions of these groups in relation to the scene without having to change the precomputed data. This allows for movement of objects and light sources while still quickly generating the correct global illumination solution for the modified scene.

In general this framework has potential for growth. Because it uses an embedded ray tracer, it should be possible to use that ray tracer to add reflective and refractive elements to the scene. The framework is also quite modular, so it should be easy to incorporate it into existing projects, making this quite attractive to companies that have an already defined system and want to incorporate an additional feature.

6.2 Limitations

Unfortunately, this algorithm has a few shortcomings. Although using a texture atlas avoids the need to divide the scene into smaller elements, there are certain complications brought upon by using textures. Even though the processing time is now proportional to the resolution of the texture, this resolution has a lower bound that depends on the number of triangles. There is also an upper bound which is the maximum texture size allowed by the hardware, currently around 4096×4096 texels. At least one pixel is needed for each triangle and in addition, larger triangles require more texture space so that harsh illumination changes like shadow boundaries are correct. At high resolutions, however, the speed decreases enormously, which is why the scenes presented in Chapter 5 have a relatively small polygon count.

Even though there is a lot of potential for expanding this algorithm, there are some lighting effects, for example caustics, that may be difficult to achieve because they require a *shooting* operation. The operations used in this framework are mainly *gather* operations, that is, for each sample point, we compute the light that is arriving there. A shooting operation requires that, for one sample point, all the other sample points affected by it must be updated, and this is hard to compute efficiently using textures on a GPU.

The implementation of the texture map also needs some work. The primary problem is that smooth surfaces look flat shaded because of the discontinuities between triangles

that are unconnected in the texture atlas but are connected in the object geometry.

6.3 Future Work

We have thought of many ideas to improve this algorithm and framework, some to reduce the current limitations and others to take advantage of the existing potential.

For aesthetic purposes only, it would be a good idea to use the concept of the temporary ambient term, proposed by the progressive refinement paper [5]. This would improve the look of the presented image while the user is waiting for the solution to be calculated.

The texture atlasing limitations can be improved by having an adaptive resolution scheme, using multiple textures or having a better atlasing scheme. Adaptive sampling can also be used, especially to calculate the textures for occluded areas since these areas usually require more resolution to look good. There are also alternatives to traditional atlasing schemes. A method called polycube maps [31] projects 3D geometry into groups of cubes and uses this projection to generate maps for seamless texturing. Another idea, geometric images [16], proposes remeshing an arbitrary surface into a completely regular structure such that it can be stored as a square image, allowing other attributes (such as texture coordinates) to be parameterized to square image. Both methods are is still at an early stage but they have potential for the kind of mapping used here.

Different scenes and even different parts of the same scene require fewer iterations to converge to a stable value. The current framework was built using a fixed number of iterations, however, an additional pass could be added that checks against a standard error value. This would stop all processing on that pixel and once all pixels reach the tolerance, the complete process would stop.

An important bottleneck is in the ray tracer, therefore any improvements that can be made to shooting the rays will greatly increase speeds. Additionally, it may be possible to reuse the information obtained from shooting the rays so that when the light sources are moved, the rays do not have to be shot again.

The mathematics used to calculate direct and indirect lighting have been well analyzed, however, there are still certain manipulations that can be done to obtain equations that have the same results but with fewer calculations. The PDF used for the direct lighting

calculation can be changed to one of uniform sampling of the solid angle subtended by the light source, as opposed to the uniform area sampling. Sampling based on the solid angle would cancel various terms from the equation, similar to what is done in the indirect illumination calculation in the next section. Another simple improvement is to use stratified sampling, in which the random samples are grouped into specific regions to ensure that they are well distributed. This might also improve coherence of ray directions, which should improve the speed of raytracing.

All these optimizations and improvements should serve to advance the framework presented here. We believe that this idea could be used to improve the interaction and quality visualization systems since it will a scene to have an illumination scheme that properly adapts to changes within it.

Appendix A

Glossary of Terms

This glossary is to help the reader understand some terms which may not be described in the main text. The definitions shown here are only in the context of this thesis and some terms may have a broader meaning than the one given.

Affine Transformation Refers to geometric transformations such as scaling, rotation and translation. In general an affine transformations preserve straight lines. This means that, if in the original geometry there existed a straight line, it will still be straight after the transformation.

Albedo Is a measure of reflectivity of a surface. In terms of illumination, an albedo is a number that describes how much percentage of light falling on a surface is reflected in a given direction. Except for special phenomena like fluorescence or phosphorescence, this number is from zero to one.

Area Light Source An area light source represents a surface of finite size that emits light. It is used to simulate ceiling lights or sunlight entering through a window. Area light sources are more difficult to simulate than point light sources because in real life an an area light source is made up of continuous flux flowing through every point in the surface.

Array An array is a collection of data in which individual items can be quickly looked up by using an indexing number.

Backend In general terms this refers to the end stage of a process. In a graphics processing system, the backend is the stage that interprets shader code and creates the adequate low level instructions for whatever graphics hardware it is configured for.

Barycentric Coordinates Barycentric coordinates are a way to represent the position of a point by specifying it in terms of three other positions. The mathematical formula is $y = wx_0 + ux_1 + (1 - uw)x_2$, where y is the point we want to represent, w and u are the barycentric coordinates that are used to weigh points x_0, x_1 and x_2.

Bi-linear Interpolation Bi-linear interpolation is used when accessing a texture to generate a smooth variation between each texel. When a texture is accessed using bi-linear interpolation, the resulting color is a weighed combination of the four nearest texels to the given texture coordinate.

BRDF Bidirectional Reflectance Distribution Function. This is a function that describes how light will be reflected on a surface. A BRDF depends on the position at the surface, the incoming angle of light and the outgoing angle (or the view angle). The function returns a reflectance value (an albedo) from zero to one (for surfaces that conserve energy) that gives the percentage of incoming light that exits at the given outgoing angle.

Buffer A buffer is a temporal data structure. In GPUs buffers are used to store textures and results of computations so that they can be used in other computations or output to the screen.

Cache Memory This is a fast but small and expensive memory located inside or very close to a processor. When a texture is accessed, a large chunk of information in the slower normal memory is stored in the cache memory and subsequent accesses to the texture are first checked in the cache for faster access.

Cache Coherency Having cache coherency means having accesses to parts of memory that are close together so that the cache is used efficiently.

Caustic Intense light caused by reflective or refractive surfaces that bend and focus light. For example, the light generated by focusing sunlight with a magnifying glass is a

caustic.

Contractive This refers to a matrix or in general any operator that, when multiplied by a variable, reduces the absolute value of that variable. In mathematical terms if T is a contractive operator, then $|Tx| < |x|$. This means that if a variable is repeatedly multiplied by a contractive operator the result will converge to zero.

Cosine Distribution Function This is a PDF that has a probability density of a cosine lobe. It is defined as $p(x) = cos(\theta)/\pi$.

Cosine Term A term that relates the projection of one vector over another.

CPU Central Processing Unit. The CPU is the part of a computer that interprets and carries out the instructions contained in the software. As opposed to the GPU of a computer, the CPU is the main processor and is better at running code with many loops and conditionals.

Energy Conserving Refers to the fact that a system that receives energy does not reflect more energy than that which was received. For example a surface receiving light energy will absorb some (which will be dissipated as heat) and reflect the remaining light.

Field of View The angle that defines how much can be seen from a viewpoint (defined by a camera or an eye). Your personal field of view is that part of the observable world that you are able to see at any given moment.

Finite Element Method A numerical technique of finding solutions for a complicated system by representing it with multiple simplified, discrete regions - i.e. finite elements.

Flux This is a fundamental radiometric quantity, also called *radiant power* that expresses how much total energy flows through a surface per unit time.

Form Factor A term in the *radiosity equation* that accounts for the geometrical relation between two patches.

Fragment Shader The part of a GPU that processes individual fragments of geometry before they become pixels. Usually, the purpose of a fragment shader is to compute the color to be applied to a fragment or to compute the depth value for the fragment or both. However, it may be used for more general purposes. The term fragment shader can also refer to a shader program that will be executed in the fragment shader unit.

Diffuse Reflection Given an irradiance distribution, a diffuse reflection occurs when the reflected radiance on a surface is independent of the exitant direction. This means that a rough or uneven surface receiving light will reflect it at many angles such that an observer will always see that surface the same, regardless of the angle.

Dot Product This is an operation between two vectors that can be used to find out the cosine relation between two vectors. It is defined as $V \cdot W = V_x W_x + V_y W_y + V_z W_z$. It can also be defined as $r_v r_w \cos(\theta)$ where θ is the angle between the vectors and r_v is the length of vector V and r_w is the length of vector W.

Expected Value Is the average or mean of a random variable.

Global Illumination Refers to the group of methods that consider the interaction of light between objects in a scene. Specifically, indirect diffuse illumination is one of the most notable effects of global illumination.

GPU Graphics Processing Unit. A specialized processor in charge of generating the visual output of a computer.

Inverse r^2 Term A term needed to account for the dispersion of power as light, or any form of energy travels from its source.

Irradiance This is the incident flux arriving on a surface, per unit surface area. It is the counterpart of Radiosity.

Iteration Iteration is the repetition of a process, or a computational procedure in which a cycle of operations is repeated, often to approximate the desired result more closely.

KD-Tree Is a space-partitioning data structure for organizing points in a k-dimensional space.

Light Source Any surface or point in space that gives off its own light.

Monte Carlo Integration A method of integrating functions by averaging a series of random samples that fall in the domain of given function.

Normal A vector that is perpendicular to a surface.

Occluder An object that blocks the view.

Perspective Projection View of a 3D design in which each element is projected to the screen along a line that intersects with the eyepoint.

Parser A parser is a computer program or a component of a program that analyses the grammatical structure of an input, with respect to a given formal grammar, a process known as parsing. In other words, it is a program that reads text and converts the text into data the computer can easily understand.

Pixel The word pixel comes from combining "picture element". It refers to the each cell that makes up the grid of a viewing screen like a monitor.

PDF Probability Distribution Function.

Pipeline A set of data processing elements connected in series, so that the output of one element is the input of the next one.

Pixel Shader See *Fragment Shader*.

Point Light Source A point light source is a representation of light that is an infinitely small point from which all light is generated. A point light source is usually used to represent candles or light bulbs. It is easy to calculate the contribution of a point light source on a surface because the illumination comes from only one direction.

Radiosity Also referred to as *Radiant Exitance*, Radiosity is the exitant flux per unit surface area. It is the counterpart of irradiance.

Radiance Flux per unit projected area per unit solid angle. In other words, the amount of power arriving to or leaving from a point on a surface, per unit solid angle and per unit projected area.

Rasterization Rasterization is the task of taking an image described in an outline format, and converting it into a series of dots for output on a grid display.

Ray Tracing Ray tracing involves calculating the results of shooting or tracing a ray from the viewing point into the scene to see what object it hits. Once the point where a surface was hit is known, local surface shading can be performed and additional rays can be traced to achieve effects like shadows, reflections and refractions.

Reflectance The percentage of light reflected by a surface. In a diffuse reflector, light is reflected equally in all directions, so the reflectance is constant, but in all other non-diffuse surfaces, a BRDF function is necessary to describe exactly how much light will be reflected in a given direction from a given incoming angle.

Rendering The process of creating an image meant to portray an object or scene, especially using computer graphics software

RenderTexture RenderTexture is a C++ class that supports the use of pbuffers for off screen rendering.

RGB Red, Green, Blue. Is a common way to represent color in computer displays. The three primary light colours, red, green and blue, are each given a value between 0 and 1. The combination of all three gives a wide range of colors, from black (all equal to 0) to white (all equal to 1).

Solid Angle A solid angle is the three dimensional analog of the ordinary angle. Instead of two lines meeting at a vertex, though, one needs a three dimensional figure that meets at a point. Simple examples of objects that do this are a cone or a pyramid. The standard unit of solid angle is the steradian which is equal to radian squared.

Shader Program A shader program is executable code used to determine the final surface properties of an object or image that is run on a GPU. This can include arbitrarily

complex descriptions of light absorption and diffusion, texture mapping, reflection and refraction, shadowing, surface displacement and post-processing effects. In addition, recent GPU architectures allow shader programs to also compute more general purpose code.

Texel Texture Element. Refers to each unit of a texture map.

Texture Map A bitmap used to texture a 3D polygon model, including adjustments for perspective correction, where vertices of the object model are mapped onto the 2D texture bitmap. In addition to color and brightness, textures may also be encoded with the properties of transparency and specular reflectivity.

Texture Atlas A complete map from a 3D structure to a 2D texture.

Texture Coordinate A coordinate that maps to a 2D image a position on the surface of a 3D object.

Tuple A variable that consist of various components.

Unit Vector A vector whose length is exactly 1.

Uniform Grid A grid that is divided into equally sized cells.

Vertex A point in 3D space with a particular location defined by x, y, and z coordinates.

Vertex Shader The section of the GPU which is in charge of executing code that modifies the geometry of the scene, as opposed to its shading.

Viewport A defined area for viewing information on a computer screen.

Vector A direction in 3D space defined by x, y, and z coordinates. A vector can be described as the difference of two vertices.

Voxel Short for volume pixel. It refers to a unit or a cell of a a three-dimensional grid.

Appendix B

Mathematical Notation

The notation used in this thesis is based on papers by Veach [32] and Kajiya [22] and on the book *Advanced Global Illumination* [9].

Vectors

An arrow over a variable indicates it is a vector, for example $\vec{v}$. A hat over a variable indicates it is a unit vector, in other words, its length is equal to one, for example $\hat{N}_x$. Vectors will always be pointing outwards from surface points, regardless of the direction of the quantity being measured.

$\vec{\omega}$, $\vec{\omega_i}$, $\vec{\omega_o}$ The vector $\vec{\omega}$ is used to represent directions. A direction can also be represented in spherical coordinates by two angles: The first angle, ϕ represents the azimuth and is measured with regard to an arbitrary axis located tangent to the surface at point x. The second angle, θ gives the elevation, measured from the normal vector N_x. In [9] directions are written as capital Greek letters Θ and Ψ. However, in papers like [32, 22] $\vec{\omega}$ is used. This notation is more common so it is the one selected for this thesis. Since vectors are, by convention, always pointing away from the surface points, the direction of the quantity being measured is defined by the subscripts i for incoming and o for outgoing.

$\vec{\ell}$, $\hat{\ell}$ Light vector. Used to represent a vector pointing *towards* a point or area light source.

$\hat{N}_x$ $\hat{N}_y$ Normals. Used to represent a vector perpendicular to a surface at a point. Normals are always assumed to be unit vectors.

Subscripts

Subscripts are used to denote specific attributes of a given variable.

i Subscript i has the widest variety of meanings. In this document it has the following uses:

- Incoming: Can be used to indicate that a vector is directed towards a surface point. Usually used with solid angle vectors. Example $\vec{\omega_i}$.
- Indirect: Indicates that a lighting quantity is produced by an integration of *indirect* lighting contributions. The special $\imath$ version of i is used to indicate an indirect component. It is usually used with radiance, irradiance or radiosity. Example $B_\imath$.
- Count or Iteration: Used to indicate an iteration of a sumation. Can be used as an indication of the iteration number or as an exponent value, in the case of the Neumann series. Example $\sum_{i=0}^{N} x_i$.

j Used only as a counter, similar to i.

e Emitted. Indicates that a lighting quantity is produced by an integration of *direct* or *self emitted* lighting contributions. Usually used with radiance, irradiance or radiosity. Example B_e.

ℓ Light source. Indicates that a lighting quantity is self emitted. In other words, it is directly emitted by a light source. Example B_ℓ. The differences between B_ℓ and B_e should be clear: B_ℓ is light generated by a light source, while B_e is direct light reflected on a surface only once.

xy From point x to point y. Used to indicate the initial and ending point of a vector. Example $\vec{\ell_{xy}}$.

Radiometric Values

The radiometric values used in this thesis are: radiance (L), radiosity (B), irradiance (E) and flux (Φ). The values can be a function based on a 3D point lying on a surface (e.i. $L(x)$) or as a general value (e.i. B).

L, $L(x)$, $L_o(x, \omega_o)$, $L_i(x, \omega_i)$ Radiance. Since radiance is bi-directional, it can be arriving at a surface point ($L_i(x, \omega_i)$) or exiting it ($L_o(x, \omega_o)$). In the case of a diffuse surface, outgoing radiance is all equal, so there is no need to write it in terms of a direction, only of position ($L(x)$).

B, B_e, $B_e(x)$, $B_i(x)$ Radiosity. Because radiosity is always exiting a surface, there is no need to use subscripts i or o (it would always be o). However, radiosity separated into two components, one for radiosity generated from direct light sources B_e and one from indirect light B_i.

E, E_e, $E_e(x)$, $E_i(x)$ Irradiance. Irradiance is similar to radiosity in that it is unidirectional, it only accounts for energy leaving a surface. Similarly, it can be separated into indirect and direct components.

Ψ Flux. This represents the power (Watts) of the light.

Miscellaneous

N, M Iteration counts. Used to represent a maximum iteration count for summations. Example $\sum_{i=0}^{N} x_i$.

A Represents total surface.

Ω Represents a hemisphere of directions.

σ Represents a solid angle. A differential solid angle is so small, it can be represented by a direction, in this case $\vec{\omega}$ is used. In equations with integrations of spheres, $d\sigma(\vec{\omega})$ will be used to represent a differential solid angle.

θ Used as the angle between a given vector or direction and the normal of the surface that the vector or direction point to or from.

T Transport operator. A radiometric value operated by T is shot in a direction and then scattered.

$f(x, \vec{\omega_i} \leftrightarrow \vec{\omega_o})$ BRDF. Will return a reflectance value ρ for a given point x for incoming radiance in direction $\vec{\omega_i}$ and outgoing direction $\vec{\omega_o}$.

Appendix C

Pseudocode Listing and Descriptions

This appendix has the pseudocode listings of functions and shader programs used in chapter 4.

C.1 Texture Atlas

C.1.1 Triangle Pairing Function

Algorithm C.1 Triangle pairing function.

```
for all unassigned T in S do
    Set T as assigned
    if there exists an unassigned triangle T' that shares an edge with T then
        Set T' as assigned
        Add pair T and pair T' to P
    else
        Add T with null pair to P
    end if
end for
Sort P from smallest pair area to largest
return P
```

Input variable S contains a listing of the triangles T within the scene. For each of these triangles, another triangle is searched that shares an edge with it. If a match is found, they are packaged into a triangle pair structure and inserted into triangle pair list P. If no unassigned adjacent triangle is found, then the triangle is inserted as a lone triangle into list P. Once all triangles have been assigned, P is sorted in order of the area size of the triangle pairs, from the smallest to largest and is returned as output (see algorithm C.1).

C.1.2 Recursive Texture Subdivision

Algorithm C.2 Recursive texture subdivision function.

if P has more than one triangle pair **then**
  $idx \leftarrow$ beginning of P
  repeat //`Find tipping point.`
    $LP \leftarrow P\,[begin]\,..P\,[idx]$
    $RP \leftarrow P\,[idx+1]\,..P\,[end]$
    $LC \leftarrow LP_{triangles}/P_{triangles}$
    $RC \leftarrow RP_{triangles}/P_{triangles}$
    $LA \leftarrow LP_{area}/P_{area}$
    $RA \leftarrow RP_{area}/P_{area}$
    $LR \leftarrow (K * LC + LA)$
    $RR \leftarrow (K * RC + RA)$
    $idx \leftarrow idx + 1$
  until $LP \geq RP$
  $Lxy \leftarrow$ lower half of xy //`Divide texture coordinates on longest axis.`
  $Rxy \leftarrow$ higher half of xy
  call Recursive Division with LP and Lxy //`Recursively divide left half.`
  call Recursive Division with RP and Rxy //`Recursively divide right half.`
else
  Set $TC \leftarrow xy$ for both triangles in pair
end if

An ordered list of triangle pairs P is given as input. If it has more than one triangle

pair it is iterated by an indexing variable idx. List P is then divided at idx point into the left side LP and right side RP. For each iteration the ratio of the number of triangles on each side to the total amount of triangles is stored in LC (left count) and RC (right count) respectively. The area ratios are similarly stored in LA (left area) and RA (right area). The ratio for amount of triangles is further weighed by a a constant K (A value of K that was found through trial and error to give "good" distributions for the test scenes was 1.5) . The ratios for each side are added as LR (left ratio) and RR (right ratio) and when the LR becomes greater than or equal to RR, it means that a balancing point has been found. After that, the input range of textures xy are divided evenly along the widest axis into Lxy and Rxy (when divided along the y axis, the lower half is considered for Lxy and the upper half for Rxy). This function is then recursively called for the left side with LP and Lxy as inputs and for the right side with RP and Rxy as inputs. If input P has only one triangle pair, then the xy coordinates are assigned to the TC list of texture coordinates for the triangles in that pair and the function exits (see algorithm C.2).

C.2 Shader Passes

C.2.1 Geometry to Texture Vertex Shader

Algorithm C.3 Geometry to texture (G2T) vertex shader.

Input x_{in} //`Vertex position.`
Input n_{in} //`Normal.`
Input tc_{in} //`Texture Coordinates.`
Input ρ_{in} //`Color value.`
$M \leftarrow$ Model Transformation Matrix
$D \leftarrow$ Device Coordinates Transformation Matrix
Output $n_{out} \leftarrow M \cdot n_{in}$
Output $x_{out} \leftarrow M \cdot x_{in}$
Output $\rho_{out} \leftarrow \rho_{in}$
Output $p_{out} \leftarrow D \cdot tc_{in}$ //`fragment position.`

The shader receives a stream of vertex positions (variable x is used for 3D positions, as it is the same notation used in all the formulas in chapters 2 and 3), normal (n), texture coordinate (tc) and color values (ρ). Additionally two matrices must be globally defined: a model transformation matrix (M) that includes any rotations, translations or scaling transformations applied to each object must be defined; and a device coordinates matrix (D) that can transform texture coordinates (ranged from $(0, 0)$ to $(1, 1)$) to device coordinates (ranged from $(-1, -1)$ to $(1, 1)$)[1]. The input values are transformed as necessary and sent as interpolated fragments to the fragment shader (see Algorithm C.3).

C.2.2 Texture to Texture Vertex Shader

Algorithm C.4 Texture to texture (T2T) vertex shader.

Input x_{in}
Input tc_{in}
Output $p_{out} \leftarrow x_{in}$ //`fragment position.`
Output $tc_{out} \leftarrow tc_{in}$

C.2.3 Geometry to 2D Image Vertex Shader

Algorithm C.5 Geometry to 2D image (G2I) vertex shader.

Input x_{in} //`Vertex position.`
Input tc_{in} //`Texture coordinates.`
$MVP \leftarrow$ Model, View and Perspective Transformation Matrix
$\Delta tc \leftarrow$ Displacement for texture coordinates //`To avoid texture bleeding.`
Output $p_{out} \leftarrow MVP \cdot x_{in}$ //`fragment position.`
Output $tc_{out} \leftarrow tc_{in} - \Delta tc$

[1] Device coordinates are those used to define positions on a computer screen or window. Even though the output will be stored in a texture, the fragment shader uses device coordinates to rasterize the image.

This shader is similar to the G2T shader but only receives a stream of vertex positions (x) and texture coordinates (tc). In addition to the information from the model transformation matrix, this shader requires knowledge of view transformations (to know the camera viewpoint) and a perspective transformation (to convert 3D points to 2D points in device coordinates). All this information is stored in matrix MVP and it is used to transform the input vertices to device coordinates. The texture coordinates also need to be displaced slightly to avoid texture bleeding as explained in section 4.2.

C.3 Radiosity Initialization Stage

C.3.1 Random Light Ray Generator (GLR) Fragment Shader

Algorithm C.6 Random light ray generator (GLR) fragment shader.

Input: x //`Surface position.`
$y \leftarrow \text{Random}(S_l)$ //`Generate a random point in the light surface.`
Output: $l_{xy} \leftarrow (y - x)$ //`Create a vector from the difference of two points.`

C.3.2 Direct Light Calculation (DL) Fragment Shader

Algorithm C.7 Direct light calculation (DL) fragment shader.

Input: N_x //`Normal.`
Input: l_{xy} //`light ray.`
Input: $V(x, y)$ //`Visibility between light point and surface point.`
$I(y) \leftarrow$ Intensity of light
$N_y \leftarrow$ Normal of light surface
$r_{xy} \leftarrow \text{length}(l_{xy})$
$l_{xy} \leftarrow \text{Normalize}(l_{xy})$
Output: $E_e(x) \leftarrow V(x, y)I(y)\frac{(N_x \cdot l_{xy})(l_{xy} \cdot N_y)}{r_{xy}^2}$

C.4 Radiosity Iteration Stage

C.4.1 Random Cosine Distribution Ray Generator (GCR) Fragment Shader

Algorithm C.8 Random cosine distribution ray generator (GCR) fragment shader.

Input: x //`Surface position`

$l_\Theta \leftarrow$ Random Cosine Distribution(Ω) //`Generate a random ray using a cosine distribution.`

$M_{Nx} \leftarrow$ Rotation Matrix // `Matrix that aligns rays to the normal of` x.

Output:$l_{xy} \leftarrow M_{Nx} \cdot l_\theta$//`Transform` l_θ `to align with the normal at` x.

C.4.2 Indirect Light Calculation (IL) Fragment Shader

Algorithm C.9 Indirect light calculation (IL) fragment shader.

Input: y //`surface point hit by ray.`

if y exists **then** //`if a surface was hit`

  Output: $E_i(x) \leftarrow B(y)$ //`return radiosity at point hit.`

else

  Output: $E_i(x) \leftarrow$ Background Color //`else return background color (usually black).`

end if

Bibliography

[1] P. Bekaert and H. Seidel. A theoretical comparison of Monte Carlo radiosity algorithms. In *Proc. 6th Fall Workshop on Vision, Modeling and Visualization*, pages 257–264, 2001.

[2] I Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *Proceedings of SIGGRAPH 2004*, pages 777–786, 2004.

[3] N Carr and John Hart. Meshed atlases for real-time procedural solid texturing. In *ACM Transactions of Graphics, Vol 21, No. 2*, pages 106–131, 2002.

[4] N Carr and John Hart. GPU algorithms for radiosity and subsurface scattering. In *Graphics Hardware*, 2003.

[5] M. Cohen, S. Chen, J. Wallace, and D. Greenberg. A progressive refinement approach for realistic image synthesis. In *Computer Graphics*, pages 75–84, 1988.

[6] M. Cohen and D. Greenberg. The hemicube: A radiosity solution for complex environments. In *Computer Graphics*, pages 31–40, 1985.

[7] M. Cohen, D. Greenberg, and D. Immel. An efficient radiosity approach for realistic image synthesis. In *IEEE Computer Graphics and Applications*, pages 26–35, 1986.

[8] G. Coombe, M. Harris, and A. Lastra. Radiosity on graphics hardware. In *Proceedings of the 2004 conference on Graphics interface*, pages 161–168, 2004.

[9] P. Dutré, P. Bekaert, and K. Bala. *Advanced Global Illumination*. AK Peters, 2003.

[10] T. Foley and J. Sugerman. KD-tree acceleration structures for a GPU ray tracer. In *Proceedings of EUROGRAPHICS on Graphics Hardware 2005*, pages 15–22, 2005.

[11] A. Fujimoto, T. Tanaka, and K. Iwata. Arts: Accelerated ray-tracing system. In *IEEE Computer Graphics and Applications, 6(4)*, pages 16–26, 1986.

[12] P. Gautron, J. Krivanek, K. Bouatouch, and S. Pattanaik. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *Eurographics Symposium on Rendering 2005*, page 102, 2005.

[13] A. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, 1989.

[14] C. Goral, K. Torrance, D. Greenberg, and B Battaile. Modeling the interaction of light between diffuse surfaces. In *Proceedings of SIGGRAPH 1984*, pages 213–222, 1984.

[15] D. Greenberg, K. Torrance, P. Shirley, J. Arvo, J. Ferwerda, S. Pattanaik, L. Lafortune, B. Walter, S. Foo, and B. Trumbore. A framework for realistic image synthesis. In *SIGGRAPH 97 Conference Proceedings*, pages 477–494, 1997.

[16] X. Gu, S. Gortler, and H. Hoppe. Geometry images. In *Proceedings of SIGGRAPH 2002*, pages 355–361, 2002.

[17] P. Hanrahan, D. Salzman, and L. Aupeperle. A radiosity hierarchical radiosity algorithm. In *Proceedings of SIGGRAPH 1991*, pages 197–206, 1991.

[18] M. Harris. Render to texture. `http://www.markmark.net/misc/rendertexture.html`.

[19] V. Havran, J. Prikryl, and W. Purgathofer. Statistical comparison of ray-shooting efficiency schemes. In *Tech Report TR-186-2-00-14 Institute of Computer Graphics, Vienna University of Technology*, 2000.

[20] H. Jensen. Global illumination using photon maps. In *Eurographics Rendering Workshop 1996*, pages 21–30, 1996.

[21] H. Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, 2001.

[22] J. Kajiya. The rendering equation. In *Proceedings of SIGGRAPH 1986*, pages 143–150, 1986.

[23] M. McCool, Z. Qin, and T. Popa. Shader metaprogramming. In *Graphics Hardware 2002*, pages 1–12, 2002.

[24] G. Moreno-Fortuny and M McCool. Unified stream processing ray tracer. In *Poster at GPGP: The ACM Workshop on General Purpose Computing on Graphics Processor, and SIGGRAPH 2004*, 2004.

[25] K. Nielsen and N. Christensen. Fast texture based form factor calculations for radiosity using graphics hardware. In *Journal of Graphics Tools 6(4)*, pages 1–12, 2002.

[26] M. Pharr and G. Humphreys. *Physically Based Rendering*. Morgan Kaufmann, 2005.

[27] T. Purcell. *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, 2004.

[28] T. Purcell, I. Buck, W. Mark, and P. Hanrahan. Ray tracing on programable graphics hardware. In *Graphics Hardware 2003*, pages 703–712, 2003.

[29] M. Sbert. Error and complexity of random walk Monte Carlo radiosity. In *IEEE Transactions on Visualization and Computer Graphics*, pages 23–38, 1997.

[30] P. Shirley. *Realistic Ray Tracing*. AK Peters, 2000.

[31] M. Tarini, K. Hormann, P. Cignoni, and C. Montani. Polycube-maps. In *Proceedings of SIGGRAPH 2004*, pages 853–860, 2004.

[32] E. Veach and L. Guibas. Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings of SIGGRAPH 1995*, pages 419–428, 1995.

[33] G. Ward, F. Rubinstein, and R. Clear. A ray tracing solution for diffuse interreflection. In *Proceedings of SIGGRAPH 1988*, pages 85–92, 1988.

[34] T. Whitted. An improved illumination model for shaded display. In *Communications of the ACM 23:6*, pages 343–349, 1980.

[35] Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/GPGPU`.