

Copyright © 2006, Oscar Mauricio Rodríguez López.
All rights reserved.

Redistributions of this document, with or without modifications, are permitted without limitations, provided that they contain the above copyright notice and this condition. Other usage is permitted without limitations of any kind.

PROGRAMMABLE SHADER ANALYSIS AND DEVELOPMENT
FOR GLOBAL ILLUMINATION RADIOSITY RENDERING

BY

OSCAR MAURICIO RODRÍGUEZ LÓPEZ

UNDERGRADUATE PROJECT

Submitted in partial fulfillment of the requirements
for the degree of Systems and Computer Engineer
in the Faculty of Engineering of the
Los Andes University

Bogotá, Colombia. May 22, 2006

PROGRAMMABLE SHADER ANALYSIS AND DEVELOPMENT
FOR GLOBAL ILLUMINATION RADIOSITY RENDERING

Oscar Mauricio Rodríguez López

Faculty of Engineering
Department of Systems and Computer Engineering
Los Andes University, 2006

Fernando de la Rosa, Advisor

Abstract

Global illumination algorithms are the pinnacle of computer graphics. The introduction of DirectX 9 in 2002 and its *programmable shaders* have brought unprecedented flexibility and power to computer graphics programmers by allowing them to write programs that alter the standard behavior of a real time computer graphics pipeline with no performance penalties.

In this work, a widely known global illumination algorithm called *Radiosity* is comprehensively analyzed, and a program that implements it using programmable shaders is developed. All details concerning the development of this program are taken into consideration, and most problems such an endeavor is likely to encounter are thoroughly analyzed. The program is then tested to measure its performance in order to determine the benefits of programmable shaders inclusion into global illumination renderers. The results obtained in this work provide important benchmarks for future developments in this area.

*To my parents Orlando and Gloria,
my brother Camilo, and Outi.
Thanks to you all, for giving me the strenght,
courage and inspiration during this whole project.*

ACKNOWLEDGMENTS

I would first like to thank my parents Orlando and Gloria, for being there all the time, for listening to my complaints and for bearing with my constant nonsense. Without their help and guidance, I would certainly never have completed this work, or at least become insane in the process. I love you!

I would also like to extend my thanks to Professor Fernando de la Rosa, for having agreed to work and bear with me for an entire year. Working with you was a great experience, and I am looking forward to make further research with you.

Special thanks to my brother Camilo, who helped me with this work, specially during the planning stage; to my closest friends Juan Camilo, Miguel and Camilo, for giving me the moral support through the development of this work; and to Priscilla for her unconditional friendship.

Thanks to the people at the Computer Graphics Research Lab IMAGINE for providing me with additional technical support; to my teachers for helping me throughout the entire project; and to the people at the Advanced Computing Center in Engineering MOX for providing me with additional software and support for this work.

Additional thanks to Microsoft for allowing me to participate in the DirectX 9 beta program and the Microsoft Visual Studio 2005 launch program, which awakened my interest in computer graphics and provided me with the necessary tools to develop this work.

CONTENTS

1. <i>Introduction</i>	1
1.1 Motivation	1
1.2 Goals of this work	1
1.3 Structure of this work	2
2. <i>Tool choice</i>	3
2.1 Are programming tools necessary?	3
2.2 Operating System	4
2.3 Graphics Engine	5
2.4 Programming Language	6
2.5 Hardware platform	7
3. <i>Programmable Shaders</i>	8
3.1 3D rendering pipelines	8
3.2 The architecture of pixel and vertex shaders	11
3.2.1 The vertex shader	12
3.2.2 The pixel shader	12
3.3 Programming shaders with HLSL	13
3.3.1 Application code	13
3.3.2 HLSL code	14
4. <i>Lighting models</i>	15
4.1 Realistic image synthesis	15
4.2 The physical model of light	15
4.3 Radiometry	16
4.4 Materials	17
4.4.1 Material color	18
4.4.2 Ambient light	18
4.4.3 The Bidirectional Reflectance Distribution Function	19
4.4.4 Diffuse surfaces	20
4.4.5 Specular surfaces	21
4.4.6 Other types of surfaces	22
4.5 Radiosity	22
4.5.1 Introduction	22

4.5.2	Considering lighting on a patch	23
4.5.3	The form factor	25
4.5.4	Putting it all together	28
5.	<i>Development and validation of the lighting methods with shaders</i>	31
5.1	Using the code	31
5.2	Loading meshes as models	32
5.3	Rendering models with Z-buffer shaders	34
5.3.1	Material color	35
5.3.2	Color and Ambient	36
5.3.3	Color, Ambient and Diffuse reflection	36
5.3.4	Color, Ambient and Specular reflection	38
5.3.5	Color, Ambient, Diffuse and Specular	39
5.4	Per-pixel lighting	41
5.4.1	General considerations on the vertex shader	41
5.4.2	Color, Ambient and Specular	42
5.4.3	Color, Ambient, Diffuse and Specular	43
5.5	Preparing models for global illumination	44
5.6	Applying radiosity	46
5.7	Enabling HDR on the application	51
5.8	Benchmark mode	53
5.8.1	Average time per vertex, changing model complexity	55
5.8.2	Average time per vertex, changing hemicube size	56
5.8.3	GPU/CPU workload ratio, changing hemicube size	56
6.	<i>Conclusions and afterthoughts</i>	58
6.1	Results	58
6.2	Motivation for future developments	58
6.3	Tool usage conclusions	58
6.4	Per-vertex illumination	59
6.5	CPU/GPU bottleneck	59
6.6	Classware	59
6.7	Future work	59
6.7.1	More GPU work	59
6.7.2	Post-processing effect shaders	60
6.7.3	Complex BRDFs	60
6.7.4	Using geometry shaders	60
6.7.5	Other global illumination algorithms	60
	<i>Appendix</i>	61
A.	<i>Demo program base code</i>	62

<i>B. Demo program mesh loading code</i>	65
<i>C. Color plates</i>	71

LIST OF FIGURES

3.1	Old style (DirectX 7) pipeline.	10
3.2	Current (DirectX 9) pipeline. Vertex and pixel processing are programmable, and results may be used as input data for more shaders in the same rendering process.	10
3.3	Future (DirectX 10) pipeline. The new geometry shaders are executed before the vertex shader.	11
4.1	Lambertian surface BRDF detail. A ray of light is summed to the final light of a patch of light, proportional to the cosine of the angle θ it forms with the patch normal \vec{N}	24
4.2	The hemicube method. A hemicube approximates a hemisphere when it's located above the patch plane.	26
4.3	Cross section of a conflicting hemicube in a convex corner. The faces that intersect the geometry will be affected by incorrect data from the outside of the shape.	26
4.4	Translated hemicube. From the new position, the hemicube won't intersect the geometry of the scene, and all incoming rays will be originated from the inside of the shape.	27
4.5	Hemicube shape problem. The irradiance of rays perpendicular to the plane affect more than other rays in order to compensate for their low density at these areas.	27
5.1	Diffuse reflection geometry. Light is pointing in the direction of $-\vec{L}_L$, but the illumination model considers vector \vec{L}_L	38
5.2	Vertex interpolation problem. Vertices at the edges are unit vectors, while interpolated vectors are not.	42
5.3	Resulting \vec{N}_v , $-\vec{N}_v$, \vec{U}_v , $-\vec{U}_v$, \vec{T}_v and $-\vec{T}_v$ calculated vectors for hemicube rendering.	49
5.4	Direction and "Up" vectors for a sample hemicube.	50
5.5	Average time per vertex graph, changing model complexity.	55
5.6	Average time per vertex graph, changing hemicube size.	56
5.7	GPU/CPU workload ratio graph, changing hemicube size.	57
B.1	Pool.Default status when two meshes are loaded	70
C.1	A simple scene, rendered using technique C. This technique runs in real time at more than 30 frames per second.	71
C.2	A simple scene, rendered using technique CA. Ambient color has obscured the entire image constantly. This technique runs in real time at more than 30 frames per second.	71

C.3	A simple scene, rendered using technique CAD. Shapes form simple lighting patterns. This technique runs in real time at more than 30 frames per second.	72
C.4	Detail of specular lighting, rendered using technique CAS. Note how per-vertex lighting forms artifacts in the floor. This technique runs in real time at more than 30 frames per second.	72
C.5	Detail of diffuse and specular lighting, rendered using technique CADS. This technique runs in real time at more than 30 frames per second.	73
C.6	Detail of specular lighting, rendered using technique CAS(P). Per-pixel lighting has solved all artifacts, and lighting is now smooth. This technique runs in real time at more than 30 frames per second.	73
C.7	Detail of diffuse and specular lighting, technique CADS(P). This technique runs in real time at more than 30 frames per second.	74
C.8	Close up of normals in the box model. Normal vectors are represented in red. . . .	74
C.9	An empty box, similar to the Cornell box, rendered using the radiosity method described in this work. This scene took 58 seconds to calculate, but is presented in real time at more than 30 frames per second.	75
C.10	A box with two shapes, radiosity rendered with an HDR light source of color (100.4, 96.8, 90.0) (almost 100 times brighter than white). The sphere projects a soft shadow on the floor, and the torus above the sphere projects a soft shadow on it. This scene took 203 seconds to calculate, but is presented in real time at more than 30 frames per second.	75

1. INTRODUCTION

1.1 Motivation

Recent developments in global illumination tend to focus on theoretical advances and mathematical structures for modelling the various components that make up light, leaving little or no room to discussions on actual implementation details. As it was learned during the development of this work, applying theory of this nature to practice is a very difficult task, for two reasons: The first and most important is that while theory is modelled by observation and experimentation on the real world and then moved to the *continuous* realm of mathematics, computers work on a *discrete* world. These two worlds differ so dramatically that certain phenomena valid in the continuous world are not valid in the discrete world, and viceversa.

The second reason is that theoretical models are developed by taking all pertinent data into account. Since computers can't consider every little detail, most computational models are developed as distant *approximations* of their theoretical counterparts.

With the latest advances in computer graphics hardware and software, it becomes crucial to analyze the implementation details of theoretical models, so that they, along with technology, can be fully taken advantage of, leaving no details unaccounted for.

1.2 Goals of this work

The main goal of this work is to comprehensively study real time programmable shaders and the basic theory of global illumination algorithms. Consequently, the radiosity global illumination method is validated by developing a radiosity renderer using HLSL shaders, whose entire development cycle is thoroughly analyzed. Certain problems are identified and solved, some by using available technology, while others by performing undocumented but important mathematical tricks. This work attempts to discuss the added functionality programmable shaders provide in order to enhance performance and graphical quality.

The information presented in this work is of great value to anyone interested in global illumination algorithms theory and implementation using programmable shaders, as well as those interested in the bleeding edge of commercial technology, where academic work rarely stands.

As another major objective, this work aims to motivate students, professors and developers into using shaders in their rendering code, and to consider global illumination into their work, by presenting topics in a simple yet thorough manner.

1.3 Structure of this work

In this work, global illumination, as well as the technology to implement it on, are studied before actually proceeding to implement the application. In chapter 2, available tools for implementing the application are discussed, and a tool set is chosen. In chapter 3, the *High Level Shader Language* HLSL is studied, as it's a very important tool when the actual implementation begins. In chapter 4, illumination models are reviewed, and the actual radiosity algorithm is studied.

With both theoretical and technological tools in hand, chapter 5 discusses the actual development cycle of a radiosity renderer, covering the most important details, and thoroughly explaining the included demo program code. With the application developed, it is tested, so conclusions can be made on chapter 6.

Appendix A describes the technical details of the demo program base code; Appendix B describes the technical details of the demo program mesh loading code; and Appendix C includes a series of full color images illustrating the methods presented in this work

In addition to this, the *Demo Program*, that illustrates the actual method used in this work is included, along with its full source code in C# and HLSL respectively. Full benchmark data is also included with this work in the companion CD.

2. TOOL CHOICE

The objective of this chapter is to describe various available technological tools in order to properly choose some to illustrate and validate the techniques presented in this work. Initially, a discussion on how choosing tools is a critical part of any academic work that illustrates techniques using source code. Then tools are classified based on their type. One by one, tools are carefully chosen, and finally, a complete set of tools is chosen for the development of this work.

2.1 *Are programming tools necessary?*

Technologies depend on technologies. Academic work is usually intended to be kept free of code, so anyone can make their own implementations on their own environments. However, when code is included, it's usually written to illustrate specific points, but even so, it is almost always tried to be kept platform-independent. Therefore, it's widely accepted that academic work, when including code, should do so by considering portability as a premium.

However, as it was learned during the development of the demo program, there is much more to developing software than just applying theory to code. Most of the theoretical material is expressed in integrals rather than sums; theoretical models are based on the real *continuous* world, while computers work on the *discrete* world, thus creating a whole lot of problems for applied theory developers. The main goal of the demo program in this work is to illustrate the various levels of shader development complexity to achieve global illumination realism.

Having said that, it's clear that a specific tool is required, and no set of choices will be perfect, for technologies always choose to improve on certain points, while leaving some other points unattended. Therefore, it's necessary to consider the necessities of the project, weigh the different possibilities, choose the best tool appropriate for the purposes of this work, and stay with it.

The main goals of the code in this work were synthesized as a defined hierarchy in the following manner:

1. **Problem illustration:** The golden rule of the code is to illustrate the various problems a developer is most likely to find, and how they are solved using shaders. This way, people who want to develop their own global illumination programs, and use this work, won't fall into the same traps the program development fell into.
2. **Efficiency:** Global illumination is slow. Any optimization, even if minimal, is greatly welcome, as it may speed up execution time by a few seconds. It's important to consider speed optimizations.

3. **Code size:** Some size optimizations were implemented. Sometimes performing the same action in multiple locations can be optimized by reusing the code, and by doing so, it may become easier to perform tests and debug the program.

It's important to notice that elements such as *portability*, *simplicity* and *maintainability* were left aside in favor of the aforementioned elements. The result demo *is not* meant to be included within other programs, as many software patterns and widely considered *good practices* were not used during the development of the demo; doing so would have restrained the development of a program that includes previously unknown elements, where a minimal change may represent a whole change of architectural design.

After considering many tools, the following set was used:

- **Operating System:** Microsoft Windows Vista Beta 2 (Build 5308).
- **Graphics Engine:** Microsoft DirectX 9.0c (February 2006 SDK). The code was also tested with Microsoft DirectX 10.
- **Platform and programming language:** C# over .NET 2.0 and HLSL 2.0
- **Hardware platform:** ATI All-In-Wonder X600 Pro. Driver: RADEON X600 Series (Microsoft Corporation - WDDM) version 7.1.4.0.

The following sections will explain the reasons for choosing these tools.

2.2 Operating System

The first task to perform when choosing tools and platforms to use is to choose an operating system or the absence of one to develop the demo in. There are two major choices: A specific operating system, and multiplatform development. Since the purpose of the application is to illustrate the problems encountered when developing a global illumination renderer, this is quite irrelevant. However, if one is to choose speed over portability, it's much easier to do so by being bound to a specific operating system. Since this is the case, the choice of binding the project to a specific operating system was made.

The next issue to address is the specific operating system to choose. The main choices are Windows, Linux and Mac. Mac was the first one to be discarded, mostly because of the lack of compatible hardware and knowledge with it. Using it might have been a good idea, but there were enough technologies to experiment with, that also experimenting with the operating system might have brought more problems than those it would have solved.

Between Windows and Linux, Windows was chosen because on Windows, all graphics engines are available, as well as binaries are generally much easier to distribute. Besides, since speed is a very important issue, it's generally a good idea to choose the system the hardware was originally developed for. Graphic adapter drivers for Windows developed by the original hardware vendor are much more common than their Linux counterparts. Because of this, Windows was chosen as the main target for the demo.

Other operating systems such as BSD, BeOS and some more, were not considered. Those operating systems are developed with other purposes different than 3D graphics, and even though they have undoubtedly good performance, the lack of original drivers also poses problems.

Having chosen Windows as the primary platform, the next issue to address is the specific version to choose. When this work started, Windows XP SP2 was used, but Windows Vista Beta 2 Build 5308 was released, and the demo was moved to Windows Vista to test the program under the future version of Windows. The benchmark, however, was run over Windows XP in order to remove any biases introduced by the operating system.

2.3 Graphics Engine

Microsoft Windows offers three major possibilities for accelerated 3D rendering: Direct3D, OpenGL and Java3D. Since speed is considered to be very important for the demo program, the possibility of integrating speed optimizations into the code was great, such optimizations are very difficult to be integrated with Java; therefore, both Java and Java3D were discarded. The main problem then was to choose between OpenGL and DirectX. This is mostly a style issue, and fanatics on either side are easy to be startled on choosing one tool over the other. The position presented by [13] follows:

OpenGL [...] isn't perfect. First of all, OpenGL has a large amount of functionality in it. Making the interface so simple requires that the implementation take care of a lot of ugly details to make sure everything works correctly. Because of the way drivers are implemented, each company that makes a 3-D card has to support the entire OpenGL feature set in order to have a fully compliant OpenGL driver. These drivers are extremely difficult to implement correctly, and the performance on equal hardware can vary wildly based on driver quality. In addition, DirectX has the added advantage of being able to move quicker to accommodate new hardware features. DirectX is controlled by Microsoft (which can be a good or bad thing, depending on your view of it) while OpenGL extensions need to be deliberated by committees.

A more modern position, presented by [16] follows:

There has been major debate and battle in deciding which API is better: Direct3D or OpenGL. On one side, Direct3D and DirectX only function on the Windows Platform, leaving little choice to cross-platform developers. However, if you are developing only on the Windows platform, it is worth considering the pros and cons of each API.

Both APIs are straightforward to use. However, DirectX usually needs more initialization code to get going. DirectX is updated more often and is better suited to take advantage of the latest technologies than OpenGL. DirectX offers a better object-oriented structure than OpenGL, but on the flipside, OpenGL is generally simpler to use. Most developers have a bias towards one or the other, but new developers can easily be satisfied with either one.

Unfortunately, a choice had to be made, and DirectX was chosen, once again, because speed optimizations would be easier to develop under Direct3D. Generally, actions can only be carried out in a single manner in OpenGL, whereas in DirectX, the same action can be carried out in many different ways, each one designed to optimize speed, memory access or readability.

Direct3D is much more complicated than OpenGL because there are many more aspects of the actions being carried out that can be customized, thus giving the developer greater control. The drawback is that hardware support must be treated very delicately, as well as extensive testing must be performed on the final product.

Direct3DX is a utility framework for Direct3D that contains many support functions which make Direct3D programming easier, with minimal performance penalties. In some aspects, it's similar to GLU, and in other aspects to GLUT for OpenGL.

The specific version chosen was DirectX 9.0c (February 2006 SDK), which was the latest version when development began. However, by the time this work was being finished, the April 2006 SDK was released. The same program should work with the new libraries, but the official library was locked to DirectX 9.0c (February 2006 SDK).

2.4 Programming Language

DirectX can be developed using a large array of programming languages. However, DirectX 9 has two different libraries: the COM and the .NET libraries. Any language that works with DLL files, can work with the COM version of the API, and most commonly, C, C++ and Delphi are used. The COM version works as close to the hardware as possible, and it's the best choice for developing light-fast applications. The .NET (also called Managed) version of DirectX can only be accessed inside .NET languages, and most commonly, Managed C++, C# and Visual Basic .NET are used. The main advantage of the .NET libraries is that system memory cleanup is performed by the .NET garbage collector, so adding or removing objects in the code is much easier, while objects located in AGP memory must be manually freed, regardless of the library being used.

Since there were many experimental features being considered, the .NET libraries were chosen, this time sacrificing speed, while valuing ease of development. This is not considered to be critical, for most of the grunt work is performed by the graphics hardware, regardless of the language.

Visual Basic .NET was the first language to be discarded. Visual Basic is a simple language with very few optimization features. Its extreme verbosity may render it inappropriate for this type of project. The other two languages considered were Managed C++ and C#, and C# was finally chosen because code produced with it tends to be cleaner than with Managed C++.

The programming language choice, however, is a style issue, and has very little relevance in the actual development of the project, specially in .NET, where the same library can be used from any of the supported languages. After that, Managed C++, C# and even J# have very similar syntax, and those familiar with one of those languages are expected to easily understand code written in any of the others.

The language the demo tool is developed in is C#, using .NET 2.0, which has many extensions to C#, many of which were widely used, such as generics and unsafe sections of code.

In DirectX, shaders can be developed in Shader Assembly or HLSL. Shader Assembly was the only choice when real-time programmable shaders were introduced in DirectX 8. By now, the *High Level Shader Language* HLSL, with a c-like syntax has widely displaced Shader Assembly programming, which is now only used to optimize shaders with specific purposes. Since this work introduces Shaders, HLSL is used instead of Shader Assembly for the sake of simplicity and clarity.

2.5 Hardware platform

Choosing hardware is the easy part. Any hardware that supports DirectX 9.0c will work. However, graphics hardware is expensive. The main testing platform available was an ATI All-In-Wonder X600 Pro, which supports DirectX 9.0c. It's not the latest and most able card, but it proved to be very useful and fast. Testing was also performed on another platform, as is shown in section 5.8.

3. PROGRAMMABLE SHADERS

The objective of this chapter is to introduce programmable shaders, explain their role and location in computer graphic applications, and analyze their structure. The *High Level Shader Language* HLSL technology is introduced, along with the method for including shaders into existing Direct3D applications.

3.1 3D rendering pipelines

The task of rendering 3D objects is usually implemented as a pipeline, where on one end, basic information for each object to be rendered, such as position, color and texturing information, are input. The pipeline then acts on this information and rasterizes the scene, which can be set to be output to either a memory buffer for more processing, or to the computer screen to present to the user.

The graphics pipeline for most modern hardware renderers can be synthesized with the following eight steps, which is a simpler model based on the models presented by StLaurent [16] and Engel [6], with the new technology of geometry shaders:

Geometry setup: The first step in a graphics pipeline is called the *geometry setup*. Abstract or concrete models are converted to a stream of vertices and triangles in this stage. Most of the times, geometry data is already setup in a way that this step is trivial, but there are many applications that require more abstract models, such as parametric models, that require a step called *tesselation*, where the surface is sampled at various points to generate a set amount of vertices and triangles that are suitable for rendering.

Some applications also store their models with very high vertex and triangle counts, so simpler models with lower vertex and triangle counts can be calculated by performing a vertex collapse algorithm. The amount of detail removed from the model is usually set to a desired level, based on target hardware (for performance issues) or as a progressive mesh technique, where models closer to the camera are drawn with a higher level of detail than those farther to it.

Geometry Shader: As the geometry is available to the program, it enters the pipeline at the *geometry shader*, which is run on a per-mesh basis, and may perform additional tessellation, or triangle collapsation, deformation and other operations at the mesh level. This shader outputs a list of vertices and triangles to draw. Geometry shaders are currently in development and haven't been deployed as the time this document is being written.

Vertex Shader: Now that the engine has a list of vertices and triangles to draw, the *vertex shader* is run on each vertex, which is able to modify the vertex properties, such as color, normal or position. Before programmable vertex shaders were available on consumer-level hardware, a fixed program was executed instead.

The vertex shader is a program run on the GPU. It receives a vertex as input, and outputs a vertex that is ready to be rendered.

Vertex post processing: Once all vertices have passed through the vertex shader, the resulting stream of vertices is then post-processed. Mainly to crop non-visible vertices. “The operations at this point include face culling, clipping, homogenous space division, and viewport considerations.” [16].

Rasterization: At this point, a stream of vertices is ready to be converted into a set of pixels ready to be output to the screen. At this point, each triangle is rasterized into a target texture, but these triangles initially have no defined colors. Instead, they have properties inherited from their parent triangle, such as Z-position and texture coordinates. These properties will be used to define the final color of the pixel, which is actually the only useful parameter for screen output.

Pixel shader: The *pixel shader* is the following step in the pipeline. A pixel shader is a program that receives the properties of a rasterized pixel, and outputs properties for the same pixel, which is usually the final color for the pixel. Some technologies call these programs *fragment shaders*, because they are processed for each fragment, or horizontal rasterized line. But in essence, both types of shaders are comparable to the point of being considered equal. Pixel shaders, however, are only run for rasterized pixels, and not for background pixels.

Pixel post processing: After each pixel has been processed, some post processing work takes place on the pixels. Effects such as fog are performed, pixel testing and alpha blending can also take place in this stage.

Display: Finally, each pixel is displayed onto the screen or a memory surface. Nothing special takes place on this stage, but the output can be stored for future use or be retrieved by the source program instead of being drawn into the screen.

When a graphics pipeline performs the steps listed above, it is said to have performed a *pass*. Complex effects are usually performed on more than one pass by having the output of one pass as an input variable on one or more other passes. The final pass usually outputs to the screen back buffer, but this is not always so, for the global illumination computations are rendered onto memory surfaces.

Old pipelines, such as the default pipeline in DirectX 9, and the available pipeline in DirectX 7 use the same model; unlike the common sense dictates, vertex and pixel shaders are available, but they are not programmable, so they aren’t usually considered as separate elements, and variables called *render states* are set to control the fixed pipeline behavior. This behavior is illustrated on figure 3.1.

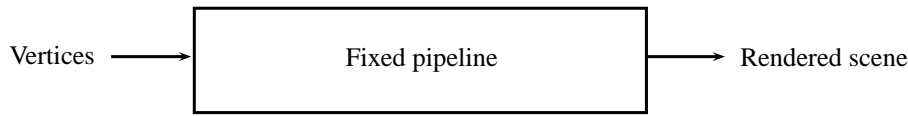


Fig. 3.1: Old style (DirectX 7) pipeline.

DirectX 8 included programmable vertex and pixel shaders, but it was until the unveiling of DirectX 9 that they became a crucial part in graphics development. A DirectX 9 application may choose to use the fixed legacy pipeline or one or more programmable vertex and pixel shaders. It's possible to use a programmable vertex shader while keeping the fixed pixel shader, but it has become very common to write both the vertex and pixel shaders as illustrated on figure 3.2. Render states are still used at the fixed parts of the pipeline. For example, choosing to rasterize triangles as solid polygons, or only their wireframes is performed by the rasterizer.

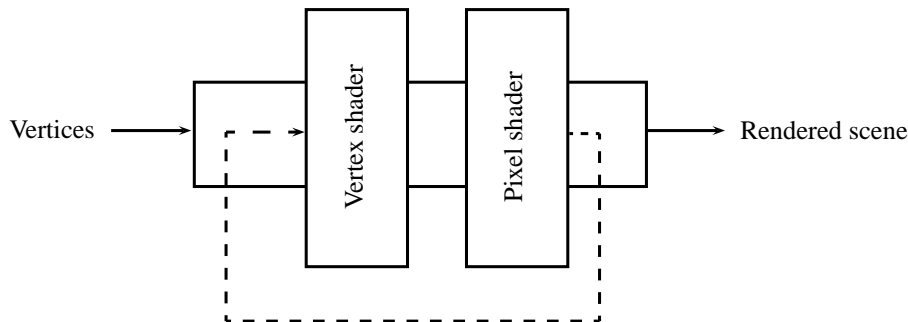


Fig. 3.2: Current (DirectX 9) pipeline. Vertex and pixel processing are programmable, and results may be used as input data for more shaders in the same rendering process.

DirectX 10, which is still being under development by the time this document is being written, offers a new version of both vertex and pixel shaders; and also introduces a new type of shader: the *Geometry Shader*, which is run before the vertex shader, and acts on the geometry, creating and deleting triangles and vertices in a programmatic manner as illustrated on figure 3.3. This development is still far from being finished, and no hardware currently exists that supports programmable geometry shaders. This work won't consider geometry shaders at all, but it's clear that being able to modify the geometry inside the pipeline will bring great advantages to computer graphics.

Pixel and vertex shaders are executed on a per-pixel and per-vertex basis respectively, regardless of the status of the rest of the pixels and vertices. Because of this, multiple shaders may be executed at the same time in order to make the entire display process much faster. As of the moment this document is being written, current consumer-level hardware supports up to 48 parallel pixel pipelines. This translates directly as that 48 pixel shaders can be run at the same time, and that it will run 48 times faster on such hardware than on single-pixel pipeline hardware.

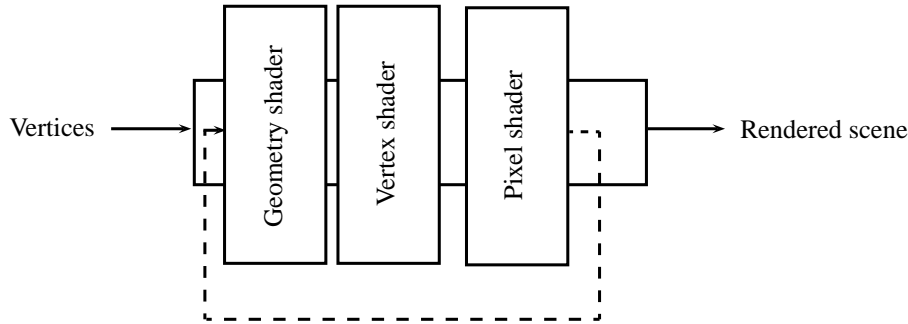


Fig. 3.3: Future (DirectX 10) pipeline. The new geometry shaders are executed before the vertex shader.

3.2 The architecture of pixel and vertex shaders

Since shaders are actually run on the graphics hardware, an architecture very different from the host computer's is found when programming shaders. This architecture is very limited, so it must be carefully studied before any considerations on the program to be developed can be made.

Both the vertex and pixel shader architectures are very similar. Based on StLaurent's explanation [16], they can be synthesized in the following manner:

Shader processor: The shader processor is the core of the shader hardware architecture. This is the element that takes the shader program and executes it.

Data registers (read only): The data registers are the input interface from the rendering pipeline. They store data specific to each object of interest to the shader, be it a vertex or a pixel, depending on the type of shader.

Constant registers (read only): Constant registers are used to pass custom information from the host program to the shader. They are constant to each render pass, but not necessarily to both the pixel and vertex shaders. Depending on the architecture, there are a fixed number of matrix, floating-point, integer and boolean registers. A very important difference between data and constant registers is that data registers have specific semantics associated with them, as they represent specific data for each vertex. Constant registers, however, contain data without any fixed semantics at all, so custom data is usually passed to the shader in constant registers.

Temporary registers (read/write): Temporary registers are used for general operations inside the shader. Local variables and temporary results are usually stored in them. On most recent hardware architectures, additional addressing registers are provided to allow loops and other control structures. Depending on the shader technology being used, these registers may become unimportant to the developer; high level shader languages such as HLSL manage them, and direct handling may only be required when the shader is to be optimized.

Output registers (write only): Output registers are the output interface to the rest of the rendering pipeline. Output data is stored on these registers.

3.2.1 The vertex shader

In HLSL version 3, the vertex shader may take one or more of the following parameters as input:

- Vertex position*
- Blend weights*
- Blend indices*
- Normal vector*
- Point size*
- Color*
- Texture coordinates*
- Tangent vector*
- Binormal vector*
- Tessellation factor*

And may output one or more of the following elements:

- Vertex position
- Point size
- Vertex fog
- Color*
- Texture coordinates*

** Denotes elements that may have more than one instance.*

3.2.2 The pixel shader

In HLSL version 3, the pixel shader may take one or more of the following parameters as input:

- Primitive face orientation (front or back)
- Pixel position
- Color*
- Texture coordinates*

Texture coordinates, as their name indicates, are generally used to store texture coordinates, however, since the shader may wish to interpret input parameters with any desired semantic, they are also used to pass custom information from the vertex shader to the pixel shader.

The pixel shader may output one or more of the following elements:

- Color*
- Depth value

* Denotes elements that may have more than one instance.

3.3 Programming shaders with HLSL

When DirectX 8 was introduced, programmable shaders were developed using Shader Assembly, an instruction-oriented programming language, that was very complicated and inflexible to use. DirectX 9 introduced the *High Level Programming Language* HLSL, a c-style syntax programming language designed specifically for programmable shaders. When HLSL 2.0 was introduced, additional elements previously unavailable such as control structures were introduced. HLSL is currently in its third installment, but since only newer hardware can run it, it is recommended for developers to try to use the lowest HLSL version a shader program supports in order to maintain backwards compatibility.

An application that uses HLSL shaders must have two different sections of code: The host application code and the shader code. The host application's role is to take care of orchestrating the shaders by loading, calling and unloading them. There is a wide array of programming languages to choose when writing the application, and it's most common to find them written in C++, managed C++, C# and VB.NET. The shader code is found on a separate file or files and it may be written using shader assembly or HLSL, which resembles C in its syntax.

3.3.1 Application code

An HLSL application has two alternatives when including shaders into its program: using *shader files* or *effect files*. The primary difference is that a shader file contains one shader, while the effect file may contain more than one shader, and meta-information that concerns the shader, such as global variables, multi-pass technique information and more. Effect files are much easier to implement, and there's an effect framework included with Direct3DX 9, a utility component included with DirectX 9. On the other side, using shader files provides greater control over the pipeline, and may be a good alternative when reusing shader code. In this work, and the demo program included with it, effect files are used.¹

For a Direct3D9 application to use effects, it must:

1. **Find the effect file:** A path to the effect file is required for the effect framework to load. This step is usually trivial.
2. **Load the effect:** The following step is to load the effect file and have the framework load it into memory. This is where most of the hard work is done by the framework. It loads and compiles the effect file, then interprets it and loads its values in memory.

This is done by calling `Effect.FromFile()`. Effects may be encoded as ASCII or binary files.

¹ HLSL effect files use the .fx extension.

3. **Set a technique:** A technique is a definition of a series of passes, where each one must contain a vertex shader and a pixel shader, and may contain an array of render states.
An HLSL effect file may contain more than one technique, and they are set by calling `Effect.Technique()`.
4. **Start the effect:** A call to `Effect.Begin()` returns the number of passes the selected technique contains.
5. **Begin a pass:** Calling `Effect.BeginPass()` will begin the a pass, as explained in section 3.1. This must be done once for each pass, and presumably in order.
6. **Render the pass:** Rendering is performed in here. Calls to `Device.DrawPrimitives()` or other primitive drawing functions, such as `Mesh.DrawSubset()` will draw primitives using the vertex shader and pixel shader selected for the current technique on the loaded effect file.
7. **End a pass:** Calling `Effect.EndPass()` will end the current pass. Steps 5 to 7 should be executed for each pass
8. **End the effect:** A call to `Effect.End()` will end the current effect. A frame has been drawn.

3.3.2 HLSL code

HLSL effect files contain all the relevant information to create, compile, execute and invoke shaders. An effect file may contain several sets of shaders, which can be selected within the code when they are used.

There are usually four sections in each effect file:

1. Global variables: Very much like C declarations, global variables may be declared at the top of the file, and their values may be changed by the host application at any time.
2. Structures: Complex structures may be declared with C-like structure syntax. They are widely used for shaders that return more than one value. e.g. Color and Position.
3. Shader code: Vertex and Pixel shaders are declared. There are no ways to identify which shaders are vertex or pixel shaders up to this point. Shaders may be written in shader assembly, as well as in one the various HLSL versions available.
4. Technique declarations: Techniques are declared. A technique may specify one or more passes, each one with a specified vertex and pixel shader, along with the language and version it is written in, so the proper compiler may be executed onto it. Render states may also be defined for each pass.

In chapter 5, various shaders are developed, and these aspects are thoroughly illustrated.

4. LIGHTING MODELS

In this chapter, lighting models are analyzed from a physical point of view. When the simplified geometrical model of light is described, the computational model of the *Bidirectional Reflection Distribution Function*, also known as BRDF is explained. The most common models of lighting are explained, and the model of *Global Illumination* is thoroughly analyzed, more specifically the *Radiosity Method*, which is then used to develop the demo program in chapter 5.

4.1 Realistic image synthesis

Rendering images as real as possible is one of the most popular goals of computer graphics. “As real as possible” does not necessarily mean an expression of reality, but rather, an image or set of images that can’t easily be discerned from the real world.

Rendering graphics is quite an extraordinary attempt. The behavior of light is very complicated, and making exact simulations is currently out of the scope of computer graphics. Instead, creating accurate approximations are the main motivation of realistic synthesis in the world, today.

Based on that, the problem of generating realistic computer graphics is based on identifying how light interacts with objects, and how that light is perceived by observers. In order to properly characterize light, it must be properly understood first.

4.2 The physical model of light

Light is a form of electromagnetic radiation, a stream of a large number of *photons*, elementary particles that travel in space. At its most fundamental state, it’s explained by *quantum mechanics* and more precisely, by *quantum optics*, which states a series of laws that are generally considered too detailed for practical use in computer graphics.

By simplifying this model, the model of light as a wave is defined. It explains certain effects, such as diffraction, interference and polarization, which can be seen frequently in the world. However, this model is also considered to be very complex, as those effects are usually faked when necessary for computer graphics.

At a more macroscopic level, geometric model of light is reached. This is a very simplified model of light, in which light is assumed to interact with objects much larger than the wavelength of light, and that the nature of light is reduced to three basic interactions: *emission*, *reflection* and *transmission*. In order to make this model as simple as possible, the following suppositions are made:

- **Direction:** Light is assumed to travel in straight lines. This rules out several effects such as diffraction, which can be faked if necessary.
- **Speed:** Light is known to travel with a certain speed c known as the *speed of light*, which is exactly $299792458m/s$. This is extremely fast for it to have a practical use in computer graphics; therefore, it's assumed that light travels instantaneously through a medium.
- **Influence:** There are certain phenomena that influence light, such as gravity and magnetic fields. Once again, they are ruled out, as these effects aren't usually relevant in computer graphics. However, they can also be faked if necessary.

Effects such as absorption and scattering are also very important, specially when rendering photorealistic images. Simulating or faking these effects is out of the scope of this work, but shaders can be used to fake them very accurately.

There's an additional issue that's very important when considering light for use in computer graphics: When considered as a wave, light is made up of a sum of several waves at with different wavelengths, while some of the mathematical models of light for use in computer graphics work only for monochromatic light. This poses a problem for color rendering. However, when used in computer graphics, colors are considered to be composed of three channels, one for red, one for green and one for blue, each one considered as a monochromatic component of light; so by performing everything three times, one for each channel, color lighting can be easily calculated with monochromatic models.

4.3 Radiometry

The goal of any lighting algorithm is to compute the interaction of light with objects, *radiometry* is the field that studies these interactions, and generally, anything concerning measuring light. The mathematical models for light used in computer graphics are based on radiometry.

The following list explains the basic quantities used in radiometry, as explained in [3] and [2]:

- **Radiant power:** *Radiant power*, also known as *flux*, is the amount of energy that flows on a surface per unit time. It's expressed in Watts (W) and is usually denoted as Φ .
- **Irradiance:** *Irradiance* is the amount of *incident* radiant power per unit surface area. It's expressed in W/m^2 and is usually denoted as E .

$$E = \frac{d\Phi}{dA} \quad (4.1)$$

- **Radiosity:** *Radiosity* is the amount of *excident* radiant power per unit surface area. It's also expressed in W/m^2 and is usually denoted as B . Some times, it's also called *Radiant excitance*, which is usually denoted as M .

$$M = B = \frac{d\Phi}{dA} \quad (4.2)$$

- **Radiance:** *Radiance* is the radiant power per unit projected area per unit solid angle. It's expressed in $W/(sr \cdot m^2)$ and is usually denoted as L .

$$L(\mathbf{x}, \vec{\omega}) = \frac{d^2\Phi}{d\omega dA^\perp} = \frac{d^2\Phi}{d\omega dA \cos\theta} \quad (4.3)$$

$$d\Phi = L(\mathbf{x}, \vec{\omega}) d\omega dA \cos\theta \quad (4.4)$$

Where \mathbf{x} is the position, and $\vec{\omega}$ is the vector of the direction being considered.

The most important property is that the radiance in the direction of a light ray remains constant as it propagates along the ray. Assuming that there are no losses due to absorption or scattering, the light ray will go from one surface to another directly, and the flux will remain constant along the ray. This means that

$$L_1 d\omega_1 dA_1 = L_2 d\omega_2 dA_2 \quad (4.5)$$

Considering that surfaces 1 and 2 are of the same size, and are parallel to each other:

$$d\omega_1 = dA_2/r_2 \text{ and } d\omega_2 = dA_1/r_1 \quad (4.6)$$

Applying 4.6 to 4.5:

$$d\omega_1 dA_1 = d\omega_2 dA_2 = \frac{dA_1 dA_2}{r_2} \quad (4.7)$$

Therefore

$$L_1 = L_2 \quad (4.8)$$

Which is also expressed as

$$L(x \rightarrow y) = L(y \rightarrow x) \quad (4.9)$$

Which means that the radiance that leaves a point equals the incoming radiance on another point, regardless of the distance between them and their geometry.

Another important property of radiance is that *sensors, such as cameras and eyes are sensitive to radiance*. This may seem quite obvious, but it states that since radiance doesn't vary with the distance between two points, the perceived brightness of a point is the same, regardless of the distance to the observer.

4.4 Materials

With the properly characterized elements that make up light, elements that interact with it are defined. In computer graphics, these elements are called materials, and they can interact in various ways with light, depending on the properties available for them on a specific rendering algorithm.

4.4.1 Material color

The first and most simple property a material has is a *color* $C_{material}$. Color is perceived as a reaction between light and the chemical properties of a material, so light may be reflected with different properties; usually, the light reflected is a subset of the sum of the incident light components, which causes white (an interference of lights with all the frequencies in the visible spectrum) objects to be seen white when white light is cast upon them, but are seen red when only red light is cast upon them. The most simple and straightforward way to render objects consists of only rendering its material color, and can be seen in action in the demo program by choosing “z-buffer” as the algorithm and “C” as the technique. The implementation details for this technique are further explained in section 5.3.1, and its results can be seen in color plate C.1.

4.4.2 Ambient light

The first type of light considered is called *ambient light*. Depending on the algorithm, it *may* be considered, and it’s very popular in local illumination algorithms.

Ambient light is a type of light that is considered to have been reflected in the world so many times that it no longer has a direction, and, for the sake of simplicity, it has the same color, regardless of the position the light is being sampled at.¹

Since this type of light comes from everywhere, the position and direction of the object mesh is irrelevant to the reflected color of the object. As mentioned in section 4.4.1, the reflected color is less than the material color, and it’s proportional to the light intensity. In general:

$$I_{ambient} = C_{ambient}C_{material} \quad (4.10)$$

Since this function works with scalars, in its native form, it can only be calculated with monochromatic lights that have simple scalar range. To convert this equation, and further lighting equations into polychromatic, it’s only necessary to calculate it once for each light component, inside arrays similar to the following matrix:

$$color = \begin{bmatrix} C_r \\ C_g \\ C_b \\ C_\alpha \end{bmatrix}_{4 \times 1} \quad (4.11)$$

Where C_r is the monochromatic red component of light, C_g is the monochromatic green component of light, C_b is the blue component of light, and C_α is called the *alpha* value, which can be used as an additional value for many different algorithms. It’s commonly seen in transparency, where the alpha value is used to determine the amount of transparency a material has; but it can also be seen in HDR ², as well as in many other effects.

¹ Notice that this assumption is very strong. If an algorithm only considers this type of light, an artifact called *bleeding*, which consists in a small but sometimes noticeable reflection on surfaces that would make a white element near a red wall have a slight red tint associated with. This is why global illumination algorithms don’t usually take this type of light into consideration.

² High Dynamic Range imaging is used when lights of different intensities are present in the same scene. For

When calculating this simple equation for each monochromatic component of light, several effects can be found. For example, if a monochromatic red object is lit with monochromatic green light:

$$C_{ambient} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}_{4 \times 1} \quad \text{and} \quad C_{material} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}_{4 \times 1} \quad (4.12)$$

By applying 4.12 to 4.10, and defining operation \otimes as a per-component multiplication of two matrices:

$$I = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (4.13)$$

Which is black. Therefore, it's very important to always keep in mind that light is calculated per component, each component being calculated monochromatically.

A sample of this model can be seen in the demo program by choosing “z-buffer” as the algorithm and “CA” as the technique. The implementation details for this technique are further explained in section 5.3.2, and its results can be seen in color plate C.2.

4.4.3 The Bidirectional Reflectance Distribution Function

Unlike ambient light, other types of lights have a direction vector. Depending on the light direction vector, the material properties, and the object direction, light will reach the object, penetrate it, probably be scattered inside it³, and leave at one or more points with specific directions. The function that defines the relation between incident and reflected radiance is called the *bidirectional scattering surface reflectance distribution function* (BSSRDF), which was first described by [12]. By assuming that there is no scattering beneath the surface, and that all light will be reflected at the same point it arrives, a simple model called the *Bidirectional Reflectance Distribution Function* (BRDF) is used [11], which is defined as the ratio between the incident light, incoming at a solid angle Ψ , exiting at a solid angle Θ , where the observer is located, at a specific point x in an object. The BRDF is intrinsic to each material, and is denoted as $f_r(x, \Psi \rightarrow \Theta)$.

$$f_r(x, \Psi \rightarrow \Theta) = \frac{dL(x \rightarrow \Theta)}{dE(x \leftarrow \Psi)} \quad (4.14)$$

$$= \frac{dL(x \rightarrow \Theta)}{L(x \leftarrow \Psi) \cos(N_x, \Psi) d\omega_\Psi} \quad (4.15)$$

example, a candle is shown in front of the sun. In this case, the display range is not enough to show both lights accurately. HDR is used to store such information, and dynamically display only a specific range that best suits all the lights in the scene

³ This is called *subsurface scattering*, which happens in most materials, and, if it's properly simulated, it yields great levels of realism on rendered images.

Where N_x is the normal at point x .

The BRDF has the following properties:

1. **Reciprocity:** The BRDF is equal when Ψ and Θ are exchanged, this means that

$$f_r(x, \Psi \rightarrow \Theta) = f_r(x, \Theta \rightarrow \Psi) \quad (4.16)$$

This principle is called the *Helmholtz reciprocity principle*, and the following notation may be used to indicate that both angles can be exchanged:

$$f_r(x, \Psi \leftrightarrow \Theta)$$

2. **Anisotropy:** The BRDF can be anisotropic. This means that if the surface is rotated around its normal, the value of f_r can change. However, there are also many isotropic materials that have an isotropic BRDF. This generally refers to the smoothness of materials. For examples, materials such as brick are clearly anisotropic; but materials such as smooth paint can be isotropic.

These two properties, and specially, reciprocity, must be carefully applied when programming BRDFs because there are some algorithms that trace light from light sources to the detector as well as algorithms that backtrace light from the detector to the light source. In order for a BRDF to be physically feasible, reciprocity is a very important property to keep into consideration.

4.4.4 Diffuse surfaces

The first type of surface to consider is perfectly diffuse. This type of surfaces are called *Lambertian surfaces*, which abide by Lambert's cosine law, that states that the light emanating from the surface is proportional to the cosine of the angle formed between the normal vector of the surface and the direction vector of the incoming light. Therefore, the brightness of the surface is equal, regardless of the viewer's position. This means that the BRDF is constant for all values of Θ .

$$f_r(x, \Psi \leftrightarrow \Theta) = \frac{C_{material}}{\pi} \quad (4.17)$$

When considering that the only light sources in a scene are those defined for the scene (not counting other objects that reflect light), the resulting intensity for a surface, with the model developed by [10] is expressed as:

$$I_{diffuse} = \sum_L F_{diffuse} C_L C_{material} \quad (4.18)$$

Where L is each light in the scene, C_L is its color, $C_{material}$ is the material color, and $F_{diffuse}$ is the diffuse factor for the specific reflection, and is defined as:

$$F_{diffuse}(L) = \vec{N} \cdot \vec{L}_L \quad (4.19)$$

With \vec{N} being the normal vector of the surface considered, and \vec{L}_L the light direction. Adding diffuse light to 4.10:

$$I_{total} = I_{ambient} + I_{diffuse} \quad (4.20)$$

$$= C_{ambient}C_{material} + C_{light}C_{material}(\vec{N} \cdot \vec{L}_L) \quad (4.21)$$

A sample of this model can be seen in the demo program by choosing “z-buffer” as the algorithm and “CAD” as the technique. The implementation details for this technique are further explained in section 5.3.3, and its results can be seen in color plate C.3.

4.4.5 Specular surfaces

The next type of surfaces to be considered are *specular surfaces*. A specular surface reflects more light in the direction of the reflected vector than other vectors. A perfect specular surface is a perfect mirror, that only reflects light in the direction of the reflected vector. In practice, simple specular surfaces have an element called *sharpness*, that indicates how much the reflected light varies as the detector changes positions. This was first modelled by Phong [14], and is called the *Phong lighting model*⁴. A reflection vector \vec{R} is calculated with the following formula:

$$\vec{R} = 2(\vec{N} \cdot \vec{\Psi})\vec{N} - \vec{\Psi} \quad (4.22)$$

Where $\vec{\Psi}$ is the light direction vector in this model.

Now, considering a scalar sharpness $n \in \mathfrak{R}$ for the surface, the resulting power is defined as:

$$I_{specular} = \cos(\vec{R}, \vec{\Theta})^n \quad (4.23)$$

Where $\vec{\Theta}$ is the direction vector from the surface to the observer. If both \vec{R} and $\vec{\Theta}$ are unit vectors, this changes to

$$I_{specular} = (\vec{R} \cdot \vec{\Theta})^n \quad (4.24)$$

Adding 4.24 to 4.10:

$$I_{total} = I_{ambient} + I_{specular} \quad (4.25)$$

$$= C_a C_m + C_s C_l (\vec{R} \cdot \vec{\Theta})^n \quad (4.26)$$

With C_a , C_m and C_l are abbreviations for $C_{ambient}$, $C_{material}$ and C_{light} respectively. C_s is the specular light component for the material.

⁴ It's important to notice that despite having the same name and creator, the Phong *lighting* model is quite different from the Phong *shading* model. In fact, the local-illumination per-pixel lighting calculations in this document use both the Phong lighting and shading models.

Where C_s is a specular reflectivity color inherent to the material. This color is usually separated from the material color $C_{material}$ to have a broader array of materials to simulate.

Samples of this model can be seen in the demo program by choosing “z-buffer” as the algorithm and “CAS” or “CAS(P)” as the technique. The implementation details for this technique are further explained in sections 5.3.4 and 5.4.2, and its results can be seen in color plates C.4 and C.6.

It’s very common to mix specular and diffuse lighting in the same scene, thus, adding 4.24 to 4.18 and 4.10, the following equation is reached, which has all of the light types mentioned up to this point:

$$I_{total} = I_{ambient} + I_{diffuse} + I_{specular} \quad (4.27)$$

$$= C_a C_m + C_i C_m (\vec{N}_L \cdot \vec{L}_L) + C_s C_l (\vec{R} \cdot \vec{\Theta})^n \quad (4.28)$$

Samples of this model can be seen in the demo program by choosing “z-buffer” as the algorithm and “CADS” or “CADS(P)” as the technique. The implementation details for this technique are further explained in sections 5.3.5 and 5.4.3, and its results can be seen in color plates C.5 and C.7.

In addition to this model, James Blinn [1] proposes an additional method to calculate specular reflection, which, according to [6], has a performance advantage. However, the unmodified Phong model runs fast enough on current hardware, so the Blinn model isn’t considered on this document.

4.4.6 Other types of surfaces

Up to this point, only two types of surfaces have been considered: diffuse and specular. Realistic rendering often requires many more types of surfaces with complex BRDFs. Since this work focuses on realistic rendering based on global illumination, rather than local illumination complex BRDFs, they won’t be considered. However, this type of surfaces may be easily integrated into global illumination scenes, there are countless works on shaders that represent those surfaces. [5, 7, 8, 9] are good sources for further information on the latest developments on this field.

4.5 Radiosity

4.5.1 Introduction

Up to this point, the only light sources in a scene being considered are those explicitly defined by the model. However, this may be considered as an incomplete model, for surfaces that have already received light may be considered as light sources for those elements that can consider them on their BRDFs. An illumination model that considers the various reflections of light on all the surfaces of the scene, as light sources, is considered to be a *global illumination model*.

The basics of radiosity rendering, in the words of Hugo Elias [4] are:

Clear your mind of anything you know about normal rendering methods. Your previous experiences may simply distract you.

I would now like to ask an expert on shadows, who will explain to you everything they know about the subject. My expert is a tiny patch of paint on the wall in front of me.

Hugo: “Why is it that you are in shadow, when a very similar patch of paint near you is in light?”

Paint: “What do you mean?”

Hugo: “How is it you know when to be in shadow, and when not to be? What do you know about shadow casting algorithms? You’re just some paint.”

Paint: “Listen mate. I don’t know what you’re talking about. My job is a simple one: any light that hits me, I scatter back.”

Hugo: “Any light?”

Paint: “Yes, any light at all. I don’t have a preference.”

So there you have it. The basic premise of Radiosity. Any light that hits a surface is reflected back into the scene. That’s any light. Not just light that’s come directly from light sources. Any light. That’s how paint in the real world thinks, and that’s how the radiosity renderer will work.

His words, although humoristic, are quite true. Radiosity rendering consists of considering all objects as potential light sources, and not only specially defined light sources. Considering the BRDF, any vector from the entire sphere surrounding a patch in a scene can contribute to its final color.

4.5.2 Considering lighting on a patch

All global illumination methods are slow compared to their local illumination counterparts. A local illumination model must, at worse, consider all light sources for all vertices on a scene. Considering L light sources, and V vertices, a local illumination’s computational complexity is $O(L \times V)$, whereas a global illumination model considers that all elements in a scene are potential light sources, therefore having computational complexity $O(V^2)$. The amount of light sources considered in a local illumination model is usually constant and much smaller than the amount of vertices by several orders of magnitude.

For example, a complex scene with 10^5 vertices and 7 light sources, as defined in local illumination models, can be rendered in real time with a specific hardware configuration at least at 30 frames per second. This means that, in rough terms, that hardware configuration has to be able to compute at least 7×10^5 triangles for each frame, which is 2.1×10^7 triangles per second. The same scene, rendered with global illumination would have to compute 10^{10} triangles, thus taking 4761.905 seconds to compute. It’s therefore clear that this problem, not even by constructing faster hardware, can be solved permanently.

Having this problem in mind, real time global illumination is ruled out, so models that take into consideration the changing position of a moving observer can’t be computed. In this light,

it's quite common for an accurate radiosity renderer to either consider that the observer is static, at least in position, and render a scene with fixed parameters; or to consider that the illumination calculations won't change as the observer moves.

The radiosity method makes the second choice, so light can be calculated beforehand, and then presented with conventional algorithms for real-time walkthroughs. In order to eliminate all elements where the observer's position is relevant, the following assumptions are made:

- **Surfaces are opaque:** Having only opaque surfaces means that only the top hemisphere centered on the surface normal is considered, while the bottom hemisphere is not. This restriction leads to consider only the reflection, and not the refraction phenomena, since refraction is dependent on the excitant light direction, which is the direction of the viewer, it's not considered.
- **Surfaces are perfect diffuse reflectors:** Depending on the actual rendering method used, some BRDFs are easier to implement than others. The ones where the excitant light direction is considered, however, are not considered. This removes the possibility for specular surfaces, where the amount of light reflected by a surface is relative to the observer's position. The BRDF considered is the one for lambertian, or diffuse surfaces, previously explained in section 4.4.4.

A lambertian surface, such as those defined in [10], have BRDFs that consider every ray of light on the top hemisphere, by summing the incoming irradiances, each one proportional to the cosine of the angle formed between its ray and the normal vector of the surface patch, as illustrated on figure 4.1, and equation 4.29:

$$E = \sum_R E_r \cos(\theta_r) \quad (4.29)$$

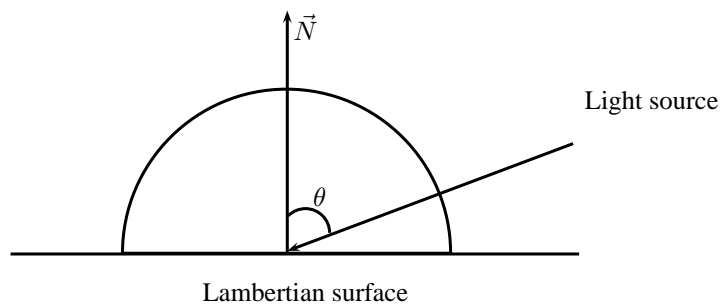


Fig. 4.1: Lambertian surface BRDF detail. A ray of light is summed to the final light of a patch of light, proportional to the cosine of the angle θ it forms with the patch normal \vec{N} .

4.5.3 The form factor

Once the computational method for light calculations is considered, another problem arises, which consists on determining which surfaces are going to light a specific patch. A solution must consider at least occlusion and distance in order to be acceptable. This problem is known as the *Form Factor Problem*, for which several approximations exist, but all can be synthesized in two great families [2]:

- **Analytic solutions** are solutions based on the geometry of objects, which are considered as whole in a scene, and require prior knowledge of the polygons involved. This type of solutions are not considered in this work.
- **Numerical solutions** are solutions where geometry is merely a restriction of the problem, but not its characterization. Many stochastic methods are quite common in this field, as well as actual hemisphere sampling, which is the method used in this work.

One of many available solutions, and the one chosen for this work consists in sampling the actual hemisphere by calculating the entire scene from each patch. If the calculation is correct, and considers occlusion and distance, it's a very accurate solution, because no special transformations widely used in stochastic methods are necessary. The precision of this solution is then directly proportional to the granularity of the hemisphere sampling, which in turn is directly proportional to computing time, therefore giving complete control over speed and precision, which can be specially tuned for each patch in the world.

Another notable advantage of hemisphere sampling over other form factor solutions is that, with very few modifications, the problem is already solved by the Z-buffer algorithm used in graphics rendering engines such as DirectX. It is then only necessary to use the engine to render the entire scene from the point of view of each patch, in the direction of the normal, with an arbitrary resolution, then apply the BRDF for the resulting pixels to obtain the final color for the patch.

This solution, however, is not perfect. The Z-buffer algorithm relies on the projection matrix to calculate the final mapping of three-dimensional objects into a two-dimensional space, which can't be used to map pixels into a hemisphere, since a sphere can't be composed of linear equations. Only with linear transformations, there are three methods to solve this problem [2]⁵:

- **Simulate a fish-eye lens at post-processing:** This is a new method, recently implementable with the inclusion of programmable shaders. It consists of rendering the scene with a wide field of view, and applying a shader that simulates the vision of a fish-eye lens at post-processing.
- **Calculate lighting for a single plane:** This method consists of rendering the scene with a wide field of view (close to π radians), then apply the BRDF for the resulting pixels, compensating for the shape of the plane.
- **Calculate lighting for a hemicube:** This method consists of rendering the scene five times: one towards the normal, and four more times in the four directions perpendicular to

⁵ Only the last two methods are explained in [2]. The first method is also included for its usefulness with shaders.

the patch plane, and orthogonal to the normal vertex. This method is illustrated on figure 4.2.

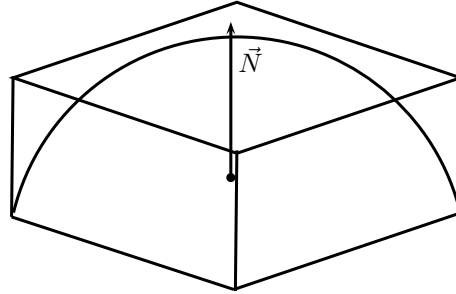


Fig. 4.2: The hemicube method. A hemicube approximates a hemisphere when it's located above the patch plane.

The hemicube method, illustrated in figure 4.2, unlike the other two, considers the whole hemisphere, thus adding to the accuracy of the result, but requires rendering the entire scene five times instead of once for each patch, effectively multiplying by five the rendering and post-processing efforts.

This method applies to any geometric model. However, when discretizing this method for graphical programming, hemicubes can be calculated for each vertex or for each polygon. This choice affects rendering times as well as graphical quality in the same way as flat shading differs from Gouraud shading. This work uses per-vertex hemicube calculations, but per-triangle calculations are very common in global illumination implementations.

Nevertheless, there are three major restrictions with the hemicube method. The first restriction is that the hemicube must have a semiside of magnitude greater than zero. The projection matrix requires the definition of a *near plane* of distance different from zero, which means that objects inside the hemicube won't be considered when the hemicube is rendered. However, the hemicube can't be set to an arbitrarily small size, because the variable types used by rendering engines are usually numerically unstable.

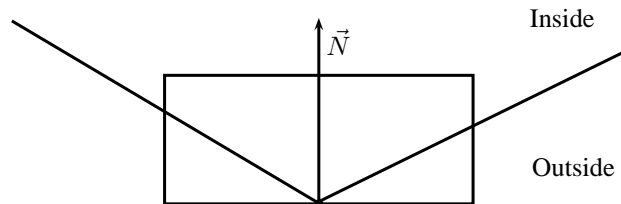


Fig. 4.3: Cross section of a conflicting hemicube in a convex corner. The faces that intersect the geometry will be affected by incorrect data from the outside of the shape.

The second restriction is also related with the hemicube size. When a hemicube intersects with the geometry of the world, artifacts that include inaccuracies in the calculation may occur because

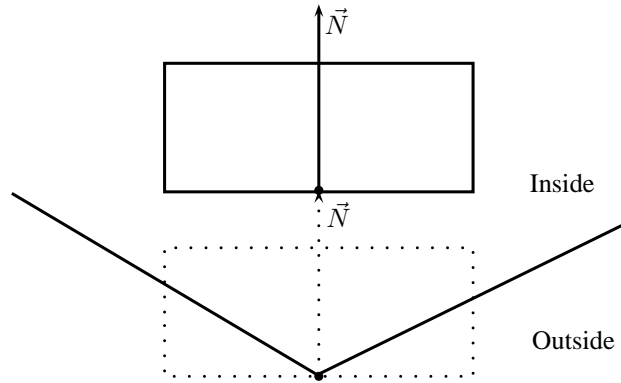


Fig. 4.4: Translated hemicube. From the new position, the hemicube won't intersect the geometry of the scene, and all incoming rays will be originated from the inside of the shape.

the Z-buffer algorithm uses a boundary representation of geometry which causes intersected triangles to be left out of the calculation, considering the geometry behind instead. This is illustrated in figure 4.3, and is a critical problem when the global illumination is calculated per-vertex instead of per-triangle, on convex corners.

This problem can be fixed by calculating the hemicube with its center a unit vector in the direction of the normal, which effectively solves the issue, with a small penalty in accuracy. This is illustrated in figure 4.4. This solution must be applied with care so the distance at which the hemicube is translated is very small compared with the dimensions of the scene. Taking this into consideration, the size of the hemicube must also be small compared with the distance at which the hemicube is translated, so this solution is worth being used.

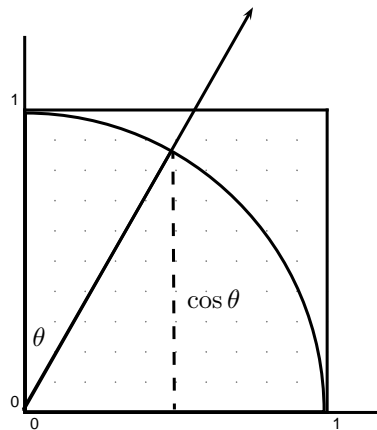


Fig. 4.5: Hemicube shape problem. The irradiance of rays perpendicular to the plane affect more than other rays in order to compensate for their low density at these areas.

The third restriction is related with the hemicube shape. A hemicube is not a sphere, and can't be considered as one so lightly. It is therefore necessary to compensate for the hemicube shape,

because it's necessary that the ray density along the hemisphere is equal for every angle, while the hemicube provides a constant ray density along the plane that makes each side of the hemicube. Failing to compensate for the shape will give greater importance to the rays in the corners of the hemicube, where more rays are considered, while central rays will have less impact on the final result.

The solution of this restriction consists in multiplying the irradiance of each ray by the cosine of the angle formed with the normal of the plane at which the ray is projected, as illustrated on figure 4.5. This way, the irradiance of incoming rays at the zenith of all planes of the hemicube will be multiplied by 1, while the irradiance of rays incoming at $\frac{\pi}{2}$ radians will be multiplied by $\frac{\sqrt{2}}{2}$, therefore correctly reducing their importance by balancing the larger density of rays at these locations.

Taking this into consideration, the total irradiance at a point in a scene is defined by equation 4.30

$$E = \sum_R E_r \cos(\theta_r) \cos(\theta_{rp}) \quad (4.30)$$

Where E_r is the irradiance for each ray, θ_r is the angle of the incoming ray, and θ_{rp} is the angle formed by the ray with the normal of its corresponding plane in the hemicube.

The radiosity for the patch is computed by multiplying the irradiance by the reflectance of the surface, given by its color, and added with the emissive component of the surface, if available.

$$B = C_{material} \sum_R E_r \cos(\theta_r) \cos(\theta_{rp}) + C_{emissive} \quad (4.31)$$

This equation is used in the demo program to calculate radiosity ⁶.

4.5.4 Putting it all together

Now that all the necessary elements for a radiosity renderer are considered, a radiosity implementation must consider the following steps:

1. **Tessellate the models to be rendered:** This is a simple yet crucial step. The radiosity method will give good approximations for the points in space it is calculated in, but the rest of the points are not. If radiosity is computed per-triangle, the scene will be rendered, and the blocks that differ in their result value will be highly noticed; if radiosity is computed per-vertex, the scene will look smoother, but locations where the tessellation resolution is not high enough to capture the change in illumination will also be noticed in the resulting image.

On the other hand, over-tessellating is also inappropriate, for sections which are evenly shaded are over-calculated, therefore wasting precious rendering time. The solution is clearly to tessellate with variable resolutions, depending on the expected variation of lighting. This problem is as complex, if not more complex than the actual global illumination problem. The

⁶ Surfaces with no reflectance are called *black bodies*, and their emissive component is usually determined by their temperature. Black bodies measured at 1200K have a red tint associated with them, while those measured at 10000K have a light blue tint.

models used in the demo program use uniform tessellation, which generate a large amount of artifacts.

2. **Locate each patch:** This is simply a matter of choosing per-triangle or per-vertex calculations. The latter is slower, as there are usually more vertices than triangles in a scene, but with better results. Per-pixel calculations are not possible using this method, because the tessellation would vary along with the observer's position. The demo program uses per-vertex calculations.
3. **Render the hemicube:** It's necessary to render the scene 5 times, once for each face of the hemicube. The results are then processed, and the BRDF and the hemicube shape compensations are applied as explained on equation 4.31.
4. **Repeat for each reflection considered:** This method considers the light from each patch, as it is computed on the patch at the moment it's considered. If all patches begin unlit (i.e. black), the first time the algorithm is executed, the radiosity generated by emissive surfaces is considered. The second time the algorithm is executed, not only the lighting generated by emissive surfaces, but also the lighting reflected on the rest of the surfaces is considered. The solution converges [4] when this algorithm is looped infinite times, but a good approximation that doesn't differ from the previous execution of the algorithm is usually achieved within 4 to 16 iterations, depending on the geometry.

The final pseudo-code for this algorithm, which was defined for this work as the *General Radiosity Algorithm*, is described in the following page.

General Radiosity Algorithm

1. Tessellate the scene into a set V of patches
2. Initialize the radiosities for every patch to “black”: $(B_{v_0} = \{0, 0, 0\} | \forall v : v \in V)$. This step consists in initializing an unlit scene.
3. Calculate lighting for the scene. This means that, considering n lighting passes, this step consists in calculating $B_{v_{i+1}}$ for all $v \in V$ vertices iteratively, for $i = (0 \dots n)$. This is accomplished with the following steps:
 - (a) Render the scene for the top plane of the hemicube using radiosities $(B_{v_i} | \forall v : v \in V)$. The resulting rendered image is stored in image E_t , which contains all incoming irradiances E_r from all directions the top plane considers.
 - (b) Render the scene for the left semiplane of the hemicube using radiosities $(B_{v_i} | \forall v : v \in V)$. The resulting rendered image is stored in image E_l , which contains all incoming irradiances E_r from all directions the left semiplane considers.
 - (c) Render the scene for the front semiplane of the hemicube using radiosities $(B_{v_i} | \forall v : v \in V)$. The resulting rendered image is stored in image E_f , which contains all incoming irradiances E_r from all directions the front semiplane considers.
 - (d) Render the scene for the right semiplane of the hemicube using radiosities $(B_{v_i} | \forall v : v \in V)$. The resulting rendered image is stored in image E_r , which contains all incoming irradiances E_r from all directions the right semiplane considers.
 - (e) Render the scene for the back semiplane of the hemicube using radiosities $(B_{v_i} | \forall v : v \in V)$. The resulting rendered image is stored in image E_b , which contains all incoming irradiances E_r from all directions the back semiplane considers.
 - (f) Calculate the resulting irradiance for the vertex, applying equation 4.30, considering the vertex material:

$$E_{v_{i+1}} = C_{material_v} \sum_R E_r \cos(\theta_r) \cos(\theta_{rp}) \quad (4.32)$$

- (g) Calculate the new radiosity for the vertex, by adding its emissive component:

$$B_{v_{i+1}} = E_{v_{i+1}} + C_{emissive_v} \quad (4.33)$$

5. DEVELOPMENT AND VALIDATION OF THE LIGHTING METHODS WITH SHADERS

This chapter discusses the development cycle of the demo program. It begins by discussing the technical details of how to use the code, so it can be easily followed by readers. It then describes each process the program considers in a descriptive and thorough manner, so every little implementation detail is covered. The objective of developing a demo program, as it was explained in chapter 1, is to consider implementation problems that theory is incapable to illustrate.

5.1 *Using the code*

The demo program was developed using C# over .NET 2.0 and DirectX 9.0c for reasons explained in chapter 2. The IDE used was Microsoft Visual Studio 2005 Version 8.0.50727.42, and a similar IDE is required to compile effortlessly the demo program.

The DirectX 9.0 SDK samples are all built on top of the DXMUT framework, which is quite similar in functionality to GLUT for OpenGL. Covering details related with window creation, event invocation, mouse/keyboard input and basic UI functionality, the DXMUT framework is a common piece of code that allows Direct3D developers to focus more on the actual details specific to the application, rather than having to code tedious elements, where bugs are very easy to appear.

A specific note on the DXMUT framework is that it uses a left-hand coordinate system, which may create several problems, and unexpected behavior may occur when the program is being executed if right hand functions are used.

The code includes a Microsoft Visual Studio 2005 project file that contains the necessary data to properly configure and build the program. It's possible, however, that the component references to the DirectX SDK are not properly resolved. If this is the case, an "unresolved reference" warning will appear. To fix this, the references must be removed and then manually added.

The external components used are:

- `Microsoft.DirectX`: The main DirectX assembly. Required for any DirectX application
- `Microsoft.DirectX.Direct3D`: The Direct3D assembly. Required for any application that uses Direct3D
- `Microsoft.DirectX.Direct3DX`: The Direct3DX support assembly. Required if its functionality is used. For the demo program, it's required.

The program was developed by adapting the "EmptyProject" sample included with the DirectX 9.0 SDK. The technical details of the pertinent modifications, as well as the information regarding

the code already included with the sample are discussed in greater detail in appendix A.

5.2 Loading meshes as models

The first problem to target on the demo program lies in how meshes and models are loaded into memory structures. It is necessary to choose, if possible, a simple and compatible file format, a coherent memory structure to hold meshes, and a set of functions to load files from the chosen format into the chosen memory structure. All three requirements are met by the Direct3DX framework with the following technologies:

- **.X Files:** X files are scalable model files for use with DirectX. They can be text encoded for greater readability, or binary encoded for greater performance and less size, with no impact on the required API to load them. .X files can contain information as simple as a single mesh, to complex animation sets. They were used to contain the following information:

- **Materials:** A .X file can contain a set of enumerable materials for later usage. These materials contain the following components¹:

- * **Material name:** A material name for further selection.
- * **Face color:** Stored as a four-component RGBA array, this color is used as the color $C_{material}$ discussed in sections 4.4.1 and 4.4.4. The C_{α} component is used for transparency, which isn't considered in this work, so it's discarded.
- * **Specular sharpness:** The specular sharpness² n discussed in section 4.4.5.
- * **Specular color:** The specular material color C_s discussed in section 4.4.5. This color is stored as a three-component RGB array.
- * **Emmissive color:** The emmissive color for this material, which is only used for global illumination algorithms, is stored as a three-component RGB array.

Once the material is defined, it can be selected into several triangles as explained below.

- **Transformation matrix:** A simple 4×4 transformation matrix is defined, which is then applied to the entire mesh. This is used to normalize all meshes to similar sizes, and to locate them as required in the scene.
- **Mesh vertices:** Mesh vertices are then defined, each one with XYZ coordinates.
- **Mesh triangles:** Triangles are defined as 0-based indexed triplets from the vertex array defined above.
- **Material correspondence:** A material is then selected into each triangle. The material count is first defined, then the triangle count, then the 0-based material index for each triangle, and finally, the material names for each index.
- **Normals:** Each vertex is then assigned with a normal vector, each defined in XYZ coordinates relative to each vertex.

¹ All colors are stored as floats, and are meant to be stored with values ranging from 0.0 and 1.0. This will be modified as HDR lighting comes into play later in section 5.7.

² Also called *specular color exponent* in the DirectX documentation for .X files.

- **Mesh** class: The **Mesh** class is the standard mesh abstraction structure in Direct3DX. It stores vertices in a vertex buffer, and triangles in various index buffers. It offers methods for locking and manipulating those vertices. It contains material structures, and divides the mesh into several *subsets*, one for each material. Rendering meshes with this class is as simple as iterating subsets, and for each one, call the **Mesh.DrawSubset()** method. Meshes are created and linked with a device, so they can be stored in video memory by creating them in **Pool.Default**, and by doing so, access can be greatly accelerated. This class also provides optimization methods for further acceleration, but these methods aren't considered in this work.
- **Mesh.FromFile()** method: The **FromFile()** method is another useful method provided by the Direct3D **Mesh** class. It provides the functionality to load **.X** files into **Mesh** objects in a single line. By using this method, loading **.X** files is left to the framework, and the developer can focus into programming the rest of the application.

Loading required meshes is performed on method **RefreshModels()**, which is called by **OnResetDevice()** on **Radiosity.cs** as well as in many other locations that request the models to be reloaded.

Meshes, however, have various limits. For instance, there is no robust way to know the current subset's material variables. There are also no fast ways to extract a single vertex without locking the vertex buffer. In order to compensate for these deficiencies, class **CompositeMesh** was created:

```
public class CompositeMesh
{
    public Mesh mesh;
    public List<Material> materialList;
    public PositionNormalDiffuseEmmissiveColorVertex[] vertices;

    public CompositeMesh()
    {
        materialList = new List<Material>();
    }
}
```

Where **mesh** is the actual mesh object, **materialList** is a list of Direct3DX Materials, and **vertices** is a copy of the vertices. Both additional elements must be filled later on when the mesh is loaded.

In addition to this, the program was developed so various models could be loaded into the same scene, so scenes could be composed with relative ease. **List<CompositeMesh> meshList** is a list that contains **CompositeMesh** objects, all of which compose the current scene.

The process of loading meshes and preparing them for global illumination is explained in detail in appendix B

5.3 Rendering models with Z-buffer shaders

Once models are properly loaded, they are ready to be rendered. A proper shader must be chosen, depending on what is to be rendered. A great advantage of using shaders is that they can be easily switched by changing `Effect.Technique`, considering that they are compatible with each other. For the following shaders, the following are defined:

```

/*****
    GLOBAL VARIABLES
*****/
float fAppTime; //App's time in seconds
float4x4 matWorld; // World matrix for object
float4x4 matWorldViewProj; // World * View * Projection matrix
float4 vecEye; // Eye vector

/*****
    CURRENT MATERIAL
*****/
float4 colorMaterialDiffuse;
float4 colorMaterialSpecular;
float fSharpness;

/*****
    CURRENT LIGHT (DIRECTIONAL)
*****/
float4 vecLightDirection;
float4 colorAmbient;
float4 colorLight;

/*****
    VS STRUCTURES
*****/
struct VS_OUTPUT_PC
{
    float4 vecPos:          POSITION;
    float4 colorResult:    COLOR0;
};

```

```
struct VS_OUTPUT_PLNV
{
    float4 vecPos:          POSITION;
    float4 vecLight:       TEXCOORD0;
    float4 vecNormal:      TEXCOORD1;
    float4 vecView:        TEXCOORD2;
};
```

5.3.1 Material color

The first shader is the trivial shader that outputs only the material color, with no lighting information, as explained in section 4.4.1. The vertex shader that does it is quite straightforward:

```
// Shader that passes on the color
VS_OUTPUT_PC VS_C( float4 vecPos : POSITION )
{
    VS_OUTPUT_PC Out;
    Out.vecPos = mul( vecPos, matWorldViewProj );
    Out.colorResult = colorMaterialDiffuse;
    return Out;
}
```

The first code line is present in all vertex shaders, and converts the vertex, which is currently in model coordinates, to projected coordinates, by multiplying it with a transformation matrix made out of the world, view and projection coordinates.

After the vertex has been converted, parameter `colorResult` is set to the passed global variable for the material color. Since this is constant for all elements of the current mesh, the Gouraud color interpolator won't have any effect, and all pixels will have the same material value.

The pixel shader is also straightforward for this, and some more shaders explained below:

```
// Trivial Shader, output our input color
float4 PS_Trivial( float4 colorResult : COLOR0 ) : COLOR0
{
    return colorResult;
}
```

This shader only takes the currently interpolated color value, and outputs it. Since all values are the same, the output image will only contain the material color for the current mesh. This shader is fast enough to be calculated at least at 30 frames per second. The color result of this shader can be seen in color plate C.1.

5.3.2 Color and Ambient

Images with only material color are somewhat plain. The first type of lighting that was implemented was ambient lighting, as discussed in section 4.4.2. The only difference between this type of lighting and using only material colors is that ambient light is multiplied by the material color. Only the vertex shader changes to accommodate to equation 4.10:

```
// Shader that calculates: Color, Ambient
VS_OUTPUT_PC VS_CA( float4 vecPos : POSITION )
{
    VS_OUTPUT_PC Out;
    Out.vecPos = mul( vecPos, matWorldViewProj );
    Out.colorResult = colorMaterialDiffuse * colorAmbient;
    return Out;
}
```

Since `colorAmbient` is constant to the scene, this will affect every mesh in it. Also, all three variables `Out.colorResult`, `colorMaterialDiffuse` and `colorAmbient` are of type `float4`, and multiplication between variables of this type is defined per-element, so the formula is properly applied. This shader is fast enough to be calculated at least at 30 frames per second. The color result of this shader can be seen in color plate C.2.

5.3.3 Color, Ambient and Diffuse reflection

The next type of lighting considered is diffuse reflection, discussed in section 4.4.4. A single directional light is considered, which affects all vertices. The vertex shader considered is the following:

```
// Shader that calculates: Color, Ambient, Diffuse
VS_OUTPUT_PC VS_CAD( float4 vecPos : POSITION,
                    float4 vecNorm : NORMAL )
{
    VS_OUTPUT_PC Out;
    float4 vecNormalWorld;
    float fDiffuse;
    float4 colorDiffuse;
    float4 vecLightWorld;
    Out.vecPos = mul( vecPos, matWorldViewProj );
    vecLightWorld =
        normalize( mul( vecLightDirection, matWorld ) );
    vecNormalWorld =
        normalize( mul( vecNorm, matWorld ) );
```

```

fDiffuse =
    saturate( dot( -vecLightWorld, vecNormalWorld ) );
colorDiffuse = colorLight * fDiffuse;
Out.colorResult = saturate(
    colorMaterialDiffuse * colorAmbient +
    colorMaterialDiffuse * colorDiffuse );
return Out;
}

```

The following elements are added:

- `float4 vecNormalWorld` contains the vertex normal \vec{N}_L for equation 4.19 in world coordinates.
- `float4 vecLightWorld` contains the light vector \vec{L}_L for equation 4.19 in world coordinates.
- `float fDiffuse` contains the diffuse reflection $F_{diffuse}(L)$ for equation 4.19. Since this value is independent of the material color, it is computed once for the entire vertex, instead of thrice for each color component.
- `float4 colorDiffuse` contains the diffuse color computed for this material, expressed by $C_{material}(\vec{N}_L \cdot \vec{L}_L)$ in equation 4.21.

The first task is to convert both the vertex and light normals, which are in model coordinates, into world coordinates. This is accomplished with a simple `mul` call. The new vectors may not be unit vectors, so they are then normalized.

Once both \vec{N}_L and \vec{L}_L are normalized, their dot product is calculated, then saturated, to obtain the diffuse factor $F_{diffuse}$. With this, the diffuse color can be computed.

It's very important however, to notice that `vecLightWorld` is actually $-\vec{L}_L$ and not \vec{L}_L ; this happens because lighting models usually consider that \vec{L}_L is pointing *out* of the surface when it is lit entirely, while the actual light vector is pointing to the opposite direction, as expressed in figure 5.1. Failing to notice this renders the scene with the inverse illumination, which might not be noticed right away, but for some specular model algorithms, such as the one explained in section 5.3.4, that use the actual light vector, diffuse reflection will be rendered in one side of the world, while specular reflection will be rendered in the other side of the world.

The diffuse illumination for the vertex can be calculated by multiplying the diffuse color of the surface, by the light color. It's this value that, when added to ambient lighting, provides the final vertex color, which can then be passed to the Gouraud interpolator to assign values to the rasterized pixels, for the pixel shader to output.

The color result of this shader can be seen in color plate C.3. This shader is fast enough to be calculated at least at 30 frames per second. Notice how the corners have different illumination. The visible artifact occurs because corners have normals pointing inside of the model, as illustrated in color plate C.8. The resulting values for each wall are different from the values in the corners, therefore having different calculated illuminations. The interpolator then expands this artifact

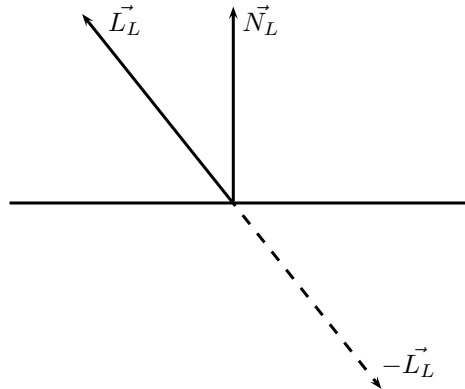


Fig. 5.1: Diffuse reflection geometry. Light is pointing in the direction of $-\vec{L}_L$, but the illumination model considers vector \vec{L}_L

through all triangles that have vertices in the corner. This problem isn't common, for the usual solution is to have different vertices in corners, with equal positions, but different normals, so illumination is calculated correctly. In global illumination, however, these vertices must have the normal pointing inside, as will be later explained in section 5.6.

5.3.4 Color, Ambient and Specular reflection

The next step is to calculate specular reflection. It's not usual to consider specular without diffuse reflection. In this case, however, for it to be thoroughly understood, it was calculated separately.

There are many methods to calculate specular reflection. The presented method is to calculate directly the method explained in section 4.4.5. The code used is the following:

```
// Shader that calculates: Color, Ambient, Specular
// Algorithm based on [6], Ch. 6
VS_OUTPUT_PC VS_CAS( float4 vecPos : POSITION,
                    float4 vecNorm : NORMAL )
{
    VS_OUTPUT_PC Out;
    float4 vecNormalWorld;
    float4 vecReflection;
    float4 vecViewDirection;
    float4 vecPosWorld;
    float fSpecular;
    float4 colorSpecular;
    float4 vecLightWorld;
    Out.vecPos = mul( vecPos, matWorldViewProj );
    vecLightWorld =
```

```

        normalize( mul( vecLightDirection, matWorld ) );
vecNormalWorld =
    normalize( mul( vecNorm, matWorld ) );
vecPosWorld = mul( vecPos, matWorld );
vecViewDirection =
    normalize( vecEye - vecPosWorld );
vecReflection =
    normalize( 2 * saturate(
        dot( vecLightWorld, vecNormalWorld ) ) *
        vecNormalWorld - vecLightWorld );
fSpecular = pow( saturate(
    dot ( vecReflection, vecViewDirection ) ),
    fSharpness );
colorSpecular = colorLight * fSpecular;
Out.colorResult = saturate(
    colorMaterialDiffuse * colorAmbient +
    colorMaterialSpecular * colorSpecular );
return Out;
}

```

The program starts in a similar way as the one that calculates diffuse reflection. The vertex position is also converted to world coordinates in `vecPosWorld` in order to calculate the view vector `vecViewDirection`, which corresponds to the excitant light vector $\vec{\Theta}$.

Following equation 4.22, the reflection vector \vec{R} is calculated and stored in `vecReflection`. Finally, the specular factor $F_{specular}$ is calculated according to equation 4.24, and the final color is composed by summing the ambient and specular colors.

The color result of this shader can be seen in color plate C.4. This shader is fast enough to be calculated at least at 30 frames per second. Notice how the corners have different illumination. The problem with low tessellation is evident in this sample, because lighting has greater variations than the actual tessellation allows for smooth coloring. Since lighting is calculated per-vertex, tessellation is directly related with result quality. There are two solutions for this problem: One involves incrementing tessellation on the models, at the expense of slower rendering, and the other is to calculate lighting per-pixel instead of per-vertex, as explained in sections 5.4.2 and 5.4.3.

5.3.5 Color, Ambient, Diffuse and Specular

Including diffuse and specular lighting in the same scene is a matter of merging both shaders in one. The code used is the following:

```

// Shader that calculates: Color, Ambient, Diffuse,
// Specular
VS_OUTPUT_PC VS_CADS( float4 vecPos : POSITION,

```

```
        float4 vecNorm : NORMAL )
{
    VS_OUTPUT_PC Out;
    float4 vecNormalWorld;
    float4 vecReflection;
    float4 vecViewDirection;
    float4 vecPosWorld;
    float fDiffuse;
    float fSpecular;
    float4 colorDiffuse;
    float4 colorSpecular;
    float4 vecLightWorld;
    Out.vecPos = mul( vecPos, matWorldViewProj );
    vecLightWorld =
        normalize( mul( vecLightDirection, matWorld ) );
    vecNormalWorld =
        normalize( mul( vecNorm, matWorld ) );
    vecPosWorld = mul( vecPos, matWorld );
    vecViewDirection =
        normalize( vecEye - vecPosWorld );
    fDiffuse =
        saturate( dot( -vecLightWorld, vecNormalWorld ) );
    vecReflection =
        normalize( 2 * saturate(
            dot( vecLightWorld, vecNormalWorld ) ) *
            vecNormalWorld - vecLightWorld );
    fSpecular = pow( saturate(
        dot( vecReflection, vecViewDirection ) ),
        fSharpness );
    colorDiffuse = colorLight * fDiffuse;
    colorSpecular = colorLight * fSpecular;
    Out.colorResult = saturate(
        colorMaterialDiffuse * colorAmbient +
        colorMaterialDiffuse * colorDiffuse +
        colorMaterialSpecular * colorSpecular );
    return Out;
}
```

The color result of this shader can be seen in color plate C.5. This shader is fast enough to be calculated at least at 30 frames per second. Notice how the corners have different illumination. The tessellation problem, however persists.

5.4 Per-pixel lighting

5.4.1 General considerations on the vertex shader

It was seen that low tessellation models can create artifacts on the final result when light is calculated per-vertex. A perfect model would not have the need for tessellation, and would properly calculate lighting for each final pixel. However, geometric definitions that eliminate the need for tessellation are too complicated and sometimes are inappropriate for fast rendering.

The solution to this problem was recently introduced with the flexibility of shaders, and consists in calculating light per-pixel without additional tessellation. The problem is that in order to calculate this properly, the vertex position and normal must be available for each pixel. A trick first presented by Phong [15] consists in interpolating the normal vector along the entire triangle and calculate lighting with these vectors. This interpolation can be performed by the Gouraud interpolator, and then lighting can be calculated per-pixel on the pixel shader.

Since programmable shaders allow the shader code to perform any interpretation on the data presented to it, and textures aren't used in this shader, various texture coordinates are used to pass the required vectors to the interpolator with the following vertex shader:

```
struct VS_OUTPUT_PLNV
{
    float4 vecPos:          POSITION;
    float4 vecLight:       TEXCOORD0;
    float4 vecNormal:      TEXCOORD1;
    float4 vecView:        TEXCOORD2;
};

// Shader that calculates vertices and passes
// them to a Phong shading pixel shader
VS_OUTPUT_PLNV VS_Phong( float4 vecPos : POSITION,
                        float4 vecNorm : NORMAL )
{
    VS_OUTPUT_PLNV Out;
    float4 vecPosWorld;
    vecPosWorld = mul( vecPos, matWorld );
    Out.vecPos =
        mul( vecPos, matWorldViewProj );
    Out.vecLight =
        normalize( mul( vecLightDirection, matWorld ) );
    Out.vecNormal =
        normalize( mul( vecNorm, matWorld ) );
    Out.vecView =
        normalize( vecEye - vecPosWorld );
}
```

```

return Out;
}

```

In this shader, semantics `TEXCOORD0`, `TEXCOORD1` and `TEXCOORD2` are used to pass the light, normal and view vectors respectively. The interpolators correctly interpolate them and pass them to the pixel shader where they can be used to calculate lighting per pixel.

Nevertheless, there are several problems with this approach which prevent this algorithm from completely displacing per-vertex calculations.

The first problem is clearly speed. With per-vertex illumination, complex calculations must only be performed on vertices, while with per-pixel illumination, they must be performed on vertices, as well as pixels. This is a great load, and considering that pixel shaders are much more complex than vertex shaders, per-pixel illumination is certain to impose a considerable performance penalty.

The second and most critical problem is that pixel shaders, being more complex than vertex shaders, have less capabilities. For specular lighting, a `pow` function is used. `pow` is only available on pixel shader 2.0 and above, thus reducing compatibility with older hardware. If per-pixel lighting is to be considered, both problems must be taken into consideration.

It's also very important to know that the interpolator is not aware that the data being passed as texture coordinates is in fact a vertex. Under this light, the interpolator will correctly interpolate each component of the vector, but the resulting vector in most cases, won't be a unit vector, as illustrated on figure 5.2. Therefore, these vectors must be normalized as soon as they enter the pixel shader if illumination is to be calculated correctly.



Fig. 5.2: Vertex interpolation problem. Vertices at the edges are unit vectors, while interpolated vectors are not.

5.4.2 Color, Ambient and Specular

Performing per-pixel calculations on the pixel shader is as simple as moving the vertex shader code to the pixel shader. The shader that calculates specular illumination on the pixel shader has the following code:

```
// Shader that calculates Color, Ambient, Specular
```

```

float4 PS_CAS( float4 vecLight:  TEXCOORD0,
               float4 vecNormal: TEXCOORD1,
               float4 vecView:  TEXCOORD2) : COLORO
{
    float4 colorResult;
    float4 vecReflection;
    float fSpecular;
    float4 colorDiffuse;
    float4 colorSpecular;
    float4 vecLightWorld = normalize ( vecLight );
    float4 vecNormalWorld = normalize ( vecNormal );
    float4 vecViewDirection = normalize ( vecView );
    vecReflection =
        normalize( 2 * saturate( dot(
            vecLightWorld, vecNormalWorld ) ) *
            vecNormalWorld - vecLightWorld );
    fSpecular = pow( saturate(
        dot ( vecReflection, vecViewDirection ) ),
        fSharpness );
    colorSpecular = colorLight * fSpecular;
    colorResult = saturate(
        colorMaterialDiffuse * colorAmbient +
        colorMaterialSpecular * colorSpecular );
    return colorResult;
}

```

The code that calculates the actual lighting is the same as the one found on the corresponding vertex shader. This code, however, specifically for the call to `pow`, requires Pixel Shader version 2.0, but as can be seen in color plate C.6, fixes the tessellation problem, without having to use a more tessellated mesh. The scene appears unlit because diffuse illumination is not considered in it. This shader is also fast enough to be calculated at least at 30 frames per second.

5.4.3 Color, Ambient, Diffuse and Specular

The final code, that integrates both diffuse and specular illumination, as well as per-pixel lighting follows:

```

// Shader that calculates Color, Ambient, Diffuse,
// Specular
float4 PS_CADS( float4 vecLight:  TEXCOORD0,
                float4 vecNormal: TEXCOORD1,
                float4 vecView:  TEXCOORD2) : COLORO

```

```

{
    float4 colorResult;
    float4 vecReflection;
    float fDiffuse;
    float fSpecular;
    float4 colorDiffuse;
    float4 colorSpecular;
    float4 vecLightWorld = normalize ( vecLight );
    float4 vecNormalWorld = normalize ( vecNormal );
    float4 vecViewDirection = normalize ( vecView );
    fDiffuse =
        saturate( dot( -vecLightWorld, vecNormalWorld ) );
    vecReflection =
        normalize( 2 * saturate( dot(
            vecLightWorld, vecNormalWorld ) ) *
            vecNormalWorld - vecLightWorld );
    fSpecular =
        pow( saturate( dot (
            vecReflection, vecViewDirection ) ), fSharpness );
    colorDiffuse = colorLight * fDiffuse;
    colorSpecular = colorLight * fSpecular;
    colorResult = saturate(
        colorMaterialDiffuse * colorAmbient +
        colorMaterialDiffuse * colorDiffuse +
        colorMaterialSpecular * colorSpecular );
    return colorResult;
}

```

The color result of this shader can be seen in color plate C.7. This shader as well, is fast enough to be calculated at least at 30 frames per second.

5.5 Preparing models for global illumination

The second part of the demo application consists in rendering scenes with global illumination. The first thing that's important to notice is that vertices contain position, normal and material data, but nothing remotely related with terms related to *Irradiance* and *Radiosity* are considered. Therefore, current models must be adapted to use this data.

Before placeholders for these variables are added, it's crucial to decide if per-triangle or per-vertex illumination is to be considered. It is under this light, that additional variables are to be added in triangle or vertex structures. Per-vertex illumination is more natural code-wise, as the vertex buffer holds information for each vertex; the only structure that describes triangles is the

index buffer, which is not fit to hold information additional to the actual vertices that make up the triangle.

On the other hand, per-vertex illumination, as seen with local illumination algorithms, tends to yield results with better graphical quality than per-triangle illumination. Per-vertex illumination assumes that light calculations vary linearly throughout the triangle. Although this is not always true, it's better than to assume that light varies in steps, as considered with per-triangle illumination. Therefore, the model considered will be calculated per-vertex. This, however, doesn't talk about vertex or pixel shaders, because the entire scene is going to be calculated once for each vertex, and this is a very complex task for it to be performed on a simple shader.

The first and most important step, is to consider incoming *Irradiance* on vertices. The actual irradiance is the final color used to present the scene to the viewer, which may be different for all vertices. Since vertices in the vertex buffer only have room for position and normal, it's necessary to make room for this additional variable. The flexible vertex format of DirectX allows a program to clone a mesh (along with its vertex buffer), specifying a new vertex format. The new vertex format consists of position, normal and "Diffuse" attributes like so:

```
public struct PositionNormalDiffuseVertex
{
    public Vector3 Position;
    public Vector3 Normal;
    public uint Diffuse;
}
```

This new `Diffuse` parameter will be passed as semantic `COLOR0` to the vertex shader. It's very important to be careful with colors packed into a single variable, for the pixel format used will greatly affect the manner in which color components are packed.

For this application, format `Format.X8R8G8B8` is selected for the front buffer, which reserves 8 bits for the red, green and blue components, while the other 8 bits are discarded. There are not many types available for display, but this format will prove to be very useful when it's used.

The second step is to make room for a copy of every vertex, each one with position, normal and color, as specified by the vertex buffer, along with its diffuse and emissive components, which will later become $C_{material}$ and $C_{emissive}$, which are specified by the corresponding material for each vertex. The mechanics of the creation of this copy are explained in appendix B.

The copy of the color component will be used to store the temporary results for the pass being computed so they can be copied after each pass in order to be ready to begin the next pass, or start presenting the scene.

The third step is to initialize the scene by setting the colors of each vertex in the vertex buffer to black, in order to have an initial dark scene. This is done when the mesh is loaded, but can also be right before the radiosity calculation is begun.

Before starting any radiosity calculations, the shader to present a colored scene is developed. The vertex shader follows:

```

struct VS_OUTPUT_PC
{
    float4 vecPos:          POSITION;
    float4 colorResult:    COLOR0;
};

// Shader that passes on the color
VS_OUTPUT_PC VS_RC(
    float4 vecPos : POSITION,
    float4 color : COLOR0 )
{
    VS_OUTPUT_PC Out;
    Out.vecPos = mul( vecPos, matWorldViewProj );
    Out.colorResult = color;
    return Out;
}

```

The only difference between this shader and the shader in section 5.3.1 is that this shader retrieves its color from the actual vertex and not from the vertex material. After that, the pixel shader is trivial:

```

// Trivial Shader, output our input color
float4 PS_RTrivial( float4 colorResult : COLOR0 ) : COLOR0
{
    return colorResult;
}

```

These shaders can be found in file `radiosity.fx` in the demo application.

5.6 Applying radiosity

Recalling the first step from the radiosity algorithm presented in section 4.5.4, it is necessary to tessellate the mesh into a set of patches. This was already accomplished when the mesh was designed, and the initial setting of the irradiance of all patches was set to black right after the mesh was loaded. The next step is to iterate for all passes. This is done in the demo program with the following code ³:

```
private void CalculateLighting(int nPasses)
```

³ This code, and some other code snippets in the radiosity calculation process don't reflect the actual code in the program, which includes additional code related with the benchmarking facilities explained in section 5.8. For additional clarity in the radiosity calculation process, this additional code was omitted.

```

{
  GenerateMaps();
  for (int i = 0; i < nPasses; i++)
  {
    CalculateRadiosity();
  }
}

```

Instead of calculating both the BRDF and the hemicube compensations for each triangle, they are precalculated on method `GenerateMaps()` by creating two multiplier maps: one for the top plane, and another for all sides of the hemicube, the first one of the size of the hemicube, and the second one half its size. They are defined in `Radiosity_Render_Radiosity.cs` as the following uninitialized bidimensional arrays of floats:

```

private float[,] MultiplierMapTop;
private float[,] MultiplierMapSide;

```

Both the BRDF and the hemicube shape compensation functions are dependent on the incoming ray's angle, but the multiplier masks are mapping functions $f'(x, y)$ over the hemicube plane. Applying basic trigonometry, it is found for the BRDF, that

$$\theta_{Top}(x, y) = \tan^{-1} \sqrt{x^2 + y^2} \quad (5.1)$$

and

$$\theta_{Side}(x, y) = \tan^{-1} \frac{\sqrt{x^2 + 1}}{-y} \quad (5.2)$$

For $0 \leq |x|, |y| \leq 1$ being the pixel position in the rendered hemicube plane.

Since angle θ_{rp} used for the hemicube shape compensation is dependent on the corresponding plane, the same method used for calculating equation 5.1 is applied, yielding

$$\theta_{rp}(x, y) = \tan^{-1} \sqrt{x^2 + y^2} \quad (5.3)$$

Composing equations 5.1, 5.2 and 5.3 with equation 4.30, the resulting mapping functions are, for the top plane:

$$M_{Top}(x, y) = \cos(\theta_{Top}) \cos(\theta_{rp}) \quad (5.4)$$

$$= \left[\cos(\tan^{-1}(\sqrt{x^2 + y^2})) \right]^2 \quad (5.5)$$

And for the side planes:

$$M_{Side}(x, y) = \cos(\theta_{Side}) \cos(\theta_{rp}) \quad (5.6)$$

$$= \cos\left(\tan^{-1} \frac{\sqrt{x^2 + 1}}{-y}\right) \cos\left(\tan^{-1} \sqrt{x^2 + y^2}\right) \quad (5.7)$$

Once the multiplier maps are created, method `CalculateRadiosity()` is called once for each lighting pass considered.

`CalculateRadiosity()` calculates radiosity for the whole scene with its current parameters in the following manner:

1. **Set the rendering target to a memory buffer of the size of the hemicycle:** This is accomplished by creating and setting a new render target of the size of the hemicycle, saving the previous render target to restore it when the calculations are complete, with the following code:

```
// Save our previous rendering target
Surface oldRenderTarget =
    device.GetRenderTarget(0);
// Create a surface to render to
newRenderTarget =
    device.CreateRenderTarget(
        RenderSize,
        RenderSize,
        Format.A8R8G8B8,
        MultiSampleType.None,
        0, true);
device.SetRenderTarget(0, newRenderTarget);
```

2. **Iterate for all vertices in the scene:** Since all vertices will be rendering points, it's necessary to iterate all vertices, first iterating over all existing meshes in the scene:

```
for (int currentViewMesh = 0;
    currentViewMesh < meshList.Count;
    currentViewMesh++)
{
    CompositeMesh CurrentViewMesh =
        meshList[currentViewMesh];
    for (int currentViewVertex = 0;
        currentViewVertex <
        CurrentViewMesh.mesh.NumberVertices;
        currentViewVertex++)
    {
        \\...
    }
}
```

}

3. **Acquire data relevant for camera positioning:** For each vertex, its position and normal are naturally obtained, which, although sufficient to render the top plane, are not enough to direct the camera to render the side planes. Up and Tangent vectors \vec{U}_v and \vec{T}_v , orthogonal between each other and with the normal vector \vec{N}_v are acquired for the vertex, by using any vector \vec{O}_v not parallel to \vec{N}_v . Since \vec{N}_v is also a unit vector, \vec{O}_v is computed like so:

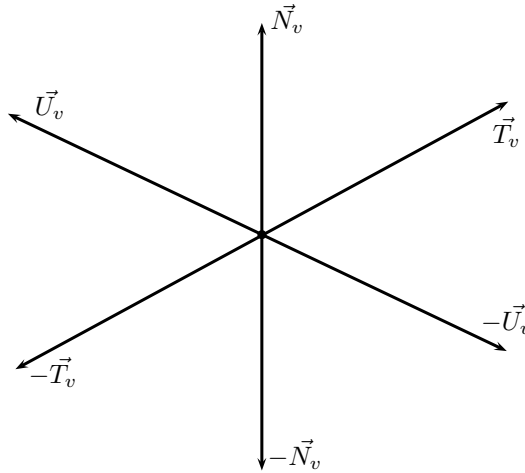


Fig. 5.3: Resulting \vec{N}_v , $-\vec{N}_v$, \vec{U}_v , $-\vec{U}_v$, \vec{T}_v and $-\vec{T}_v$ calculated vectors for hemicube rendering.

$$\vec{O}_v = \begin{cases} (0, 1, 0) & \text{if } \vec{N}_v \approx [1, 0, 0] \vee \vec{N}_v \approx [-1, 0, 0] \\ (1, 0, 0) & \text{otherwise} \end{cases} \quad (5.8)$$

By using the orthogonality property of the cross product, \vec{U}_v and \vec{T}_v are calculated in the following manner:

$$U_v = O_v \times N_v \quad (5.9)$$

$$T_v = U_v \times N_v \quad (5.10)$$

Figure 5.3 illustrates the resulting vectors. In addition to this, the position vector is moved by a unit along \vec{N} , in order to avoid the problem presented in section 4.5.3.

4. **Render the hemicube:** By calling `RenderRadiosityCalculation()`, the entire scene is rendered at a specific point, with direction and “up” vectors for the camera. For a vertex positioned at P_v , this method is called five times with the attributes illustrated in table 5.1, and figure 5.4:

It’s desired to keep all “up” vectors as \vec{N}_v for the side planes. This way, only the top half of the rendered image is considered, thus being consistent along all the side planes.

Position	Direction	“Up” vector	Hemicube plane
\vec{P}_v	\vec{N}_v	\vec{U}_v	Top plane
\vec{P}_v	\vec{U}_v	\vec{N}_v	Left side plane
\vec{P}_v	$-\vec{U}_v$	\vec{N}_v	Front side plane
\vec{P}_v	\vec{T}_v	\vec{N}_v	Right side plane
\vec{P}_v	$-\vec{T}_v$	\vec{N}_v	Left side plane

Tab. 5.1: Camera attributes for each hemicube rendering.

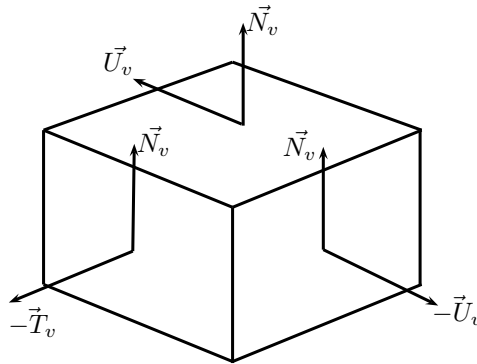


Fig. 5.4: Direction and “Up” vectors for a sample hemicube.

The pixel shader used to render the hemicube considers the color of the current vertex material by multiplying it with the resulting color for each pixel. This material color *is not* the material of the vertex being rendered, but the one of the vertex whose lighting is being calculated. This shader, however, is considered with greater detail in section 5.7.

5. **Sum the results:** Once all planes are rendered, the rendering targets are examined. The total irradiance for the hemicube is computed by adding the value of each pixel multiplied by its corresponding value in its multiplier map. The new radiosity for the vertex is computed as the sum between the total irradiance and the emissive component of the vertex material. It is then saturated, encoded in X8R8G8B8 format as explained in section 5.7 and stored as the new color value of the vertex, but not yet inside the vertex buffer.
6. **Update the vertex buffer:** After all vertices have been calculated, the `Diffuse` parameter of all vertices inside the vertex buffer are updated with their calculated radiosities. This way, they are ready for either the next pass or presentation.
7. **Return to the original render target:** The final step is to return to the original render target, stored at the beginning of the pass. The render target used for the hemicube, as well as the copy created from the original render target are then disposed of. A new pass can then begin.

This process yields very interesting results. The color results of this method can be seen on color plates C.9 and C.10. These results, however, come at a cost. The test computer took 58 seconds

and 203 seconds respectively to calculate them.

5.7 Enabling HDR on the application

[16] gives an excellent introduction to the topic:

“There have been many debates in the past about whether eight bits of resolution per color component are enough to represent all the colors the human eye can see. Generally speaking, eight bits are plenty, but this does not take into consideration something that is of crucial importance in photograph.

Although 256 colors per component is sufficient to represent color, what about light intensity? The difference in intensity between candlelight and sunlight is on the order of thousands to one, which obviously can’t be represented with only eight bits worth of resolution. You may think this is of little importance, but here are a few real-life examples of how this makes a difference.

Let’s say you are outside in the sunlight looking into a torch-lit cave. From this position, you most likely cannot see anything inside the cave, and it appears dark. If you flip the situation and you are inside the cave looking out, you can see the inside of the cave fine, but the outside appears very bright, and you cannot see much of what is out there. The reason behind this phenomenon is that both cameras and the human eye can only see a certain range of brightness, and both have exposure control mechanisms, which control the amount of light that can be seen based on the average light of their surroundings.

[...]

As you can see [...], taking light intensity into account can be a strong factor in creating more realistic and immersive renderings. Because of this, many studies have been done on the phenomenon called *high dynamic range*, or HDR.”

HDR is often used for effects [16] such as Glare, Streak and Lens flares. However, the need for HDR arises in global illumination as light sources, and more specifically, emissive lights are much brighter than reflective surfaces. If only the color, and no brightness is considered, a the radiosity of a patch may overwhelm the one of the light source, and the calculation, although correct, may not have the desired effect.

The problems arising with the lack of HDR in a global illumination algorithm are characterized in the following manner:

- **Lack of brightness:** Considering only a small light source in a scene, the rendered hemicube on a patch across the room will consider the light source, but it’s so small that will affect very little in the resulting total irradiance for the patch, so it will be very dimly lit, and the final scene, regardless of the amount of lighting steps used, will be extremely dark. This problem can be solved computationally by considering that it’s possible to have:

$$\sum B > \sum E \tag{5.11}$$

This effect, computationally called *overexposure*, breaks the second law of thermodynamics by multiplying the total radiance by a constant, which effectively solves this problem.

- **Patch overwhelming:** The second problem is that, without high range lighting, all surfaces will yield similar values at the moment of computing irradiance on a patch. Patches will then be affected by a light source in the same manner as reflecting surfaces, and only scenes with very low illumination brightnesses can be simulated. For example, a scene with a white light source in the middle of an already lit red wall will consider the redness of the wall in the same manner as the white light source, and the resulting irradiance for a patch across the room with both the light source and the wall in its field of view will be computed as a dark pink, instead of an expected very light pink.

Noth problems can be solved with HDR. The most common way to treat HDR is with a pixel format with greater precision, such as `A2R10G10B10`, or even `A16B16G16R16` if the target hardware supports it. These formats are used to encode images with HDR, but any HDR calculations must be performed at the shader level, which outputs data exclusively in `A8R8G8B8`.

The programmable shader infrastructure allows developers to pass data into the shader with many different types, including a high precision `float`, so input of high range colors is not a problem; but, part of the radiosity calculations are performed on the program outside of the shader, so it's necessary to have high range output, which is not possible in a high precision pixel format.

The solution to HDR with 8-bit precision is was first introduced by [17], and consists in using the alpha component for HDR in a format called `E8R8G8B8`, where the `E` stands for *exponent*. Since this component is often used for transparency effects, using it for HDR rules out transparency in the scene.

Encoding colors in the `E8R8G8B8` format consists in dividing the actual color value by a value called *range*, which is greater or equal than all red, green and blue components. This way, all components are normalized, and then stored with 8-bit precision. The range is then stored in the `E` component, but usually not as is, but its base 2 logarithm. This encoding format causes some loss of precision, which is usually acceptable, because a small change in high range lighting is not noticeable. Decoding `E8R8G8B8` values is as simple as retrieving the range, and multiplying all components by it.

With high range lighting, it's possible to have a light source 10 times as bright as the standard range. Encoding is performed in the pixel shader of the radiosity calculation with the following code:

```
float4 valueEncoded;
float maxComponent =
    max( max( valueResult.r, valueResult.g ),
        valueResult.b );
float fExponent = ceil( log2( maxComponent ) );
valueEncoded.rgb = valueResult.rgb / exp2(fExponent);
valueEncoded.a = (fExponent + 128) / 255;
```

```
return valueEncoded;
```

Variable `maxComponent` stores the maximum between the red, green and blue components, and its base 2 logarithm is calculated and rounded to the next integer. The red, green and blue components are divided by the range, which is `exp2(fExponent)`. The exponent is stored in the alpha component as $\frac{E+128}{255}$ to allow ranges lesser than 0, yielding a minimum range of $[0 \dots 2^{-128}]$ and a maximum range of $[0 \dots 2^{127}]$.

Encoded pixels must then be decoded by the program when the incoming irradiance is computed. This is achieved with the following code, executed for each pixel:

```
encodedR =
    ((float)data[j * RenderSize + i].R);
encodedG =
    ((float)data[j * RenderSize + i].G);
encodedB =
    ((float)data[j * RenderSize + i].B);
encodedE =
    ((float)data[j * RenderSize + i].E) / 255.0f;
fRange = (float)(Math.Pow(2.0,
    (encodedE * 255.0f) - 128.0f));
decodedR = encodedR * fRange;
decodedG = encodedG * fRange;
decodedB = encodedB * fRange;
```

With the decoder in place, HDR lighting is enabled on the scene, and light emitters are no longer restricted to the $[0 \dots 1]$ range.

The complete radiosity rendered scene with HDR can be seen in color plates C.9 and C.10.

5.8 Benchmark mode

Radiosity rendering is slow. In order to measure how the running speed of the actual program varies when rendering parameters and scene complexity are variable, an additional mode called *benchmark mode* was created. The benchmark mode calls the radiosity algorithm several times with different parameters, and uses time measuring functions located in several parts of the algorithm. The final goal is to measure the most important variables, which were defined as the following:

- How much calculation time is affected by model complexity and computer hardware
- How much calculation time is affected by hemicube size and computer hardware
- How computing time is distributed between the CPU and GPU and computer hardware

In order to accomplish this, the following variables are measured:

- Time required to render the hemicube (Executed in the GPU)
- Time required to calculate the irradiance based on the hemicube (Executed in the CPU)
- Total computing time

The benchmark executes the following tests:

- **Test 1: Vary the model complexity, rendering with 64 pixel hemicubes.** The same model is used, but with different tessellation factors.
 1. Tessellation factor 20: 774 vertices, 1200 triangles, 10 passes.
 2. Tessellation factor 10: 2734 vertices, 4800 triangles, 10 passes.
 3. Tessellation factor 5: 10254 vertices, 19200 triangles, 10 passes.
 4. Tessellation factor 4: 15814 vertices, 30000 triangles, 10 passes.
- **Test 2: Vary the hemicube size, rendering with the simple (774 vertices, 1200 triangles) scene.** The same model is used with a fixed tessellation, varying only the hemicube size.
 1. Hemicube size: 8 pixels. 5 passes.
 2. Hemicube size: 16 pixels. 5 passes.
 3. Hemicube size: 64 pixels. 5 passes.
 4. Hemicube size: 128 pixels. 5 passes.
 5. Hemicube size: 256 pixels. 5 passes.
 6. Hemicube size: 512 pixels. 5 passes.

The benchmark was executed on two different platforms:

- **Platform 1: “Outi”**
 - **Processor:** Intel Pentium 4 HT 530 (Prescott) 3.00GHz, 1024KB L2 cache (1 physical unit, 1 logical unit)
 - **System board:** Intel D915GEV, i915P/i915G chipset
 - **Memory:** Corsair DDR2-SDRAM, 1024MB
 - **Display adapter:** ATI Technologies All-In-Wonder X600 Series, PCIE 16X, 400MHz Internal DAC, 256MB
 - **Operating system:** Microsoft Windows XP Professional (5.1, Build 2600)
 - **DirectX Version:** DirectX 9.0c (4.09.0000.0904)
- **Platform 2: “Girardot”**
 - **Processor:** 2×Intel Xeon (Prestonia) 2.66GHz, 512KB L2 cache (2 physical units, 2 logical units)

- **System board:** Dell 0F1263 Intel E7505 chipset
- **Memory:** Samsung DDR-SDRAM, 2048MB
- **Display adapter:** nVidia Quadro FX 3000, AGP 8X, 400MHz Internal DAC, 256MB
- **Operating system:** Microsoft Windows XP Professional (5.1, Build 2600)
- **DirectX Version:** DirectX 9.0c (4.09.0000.0904)

Based on these tests⁴, the following comparisons can be made:

Note: All times are expressed in 100-nanosecond intervals

5.8.1 Average time per vertex, changing model complexity

This test is used to determine how the model size affects the total computing time, and can be deduced entirely from test 1.

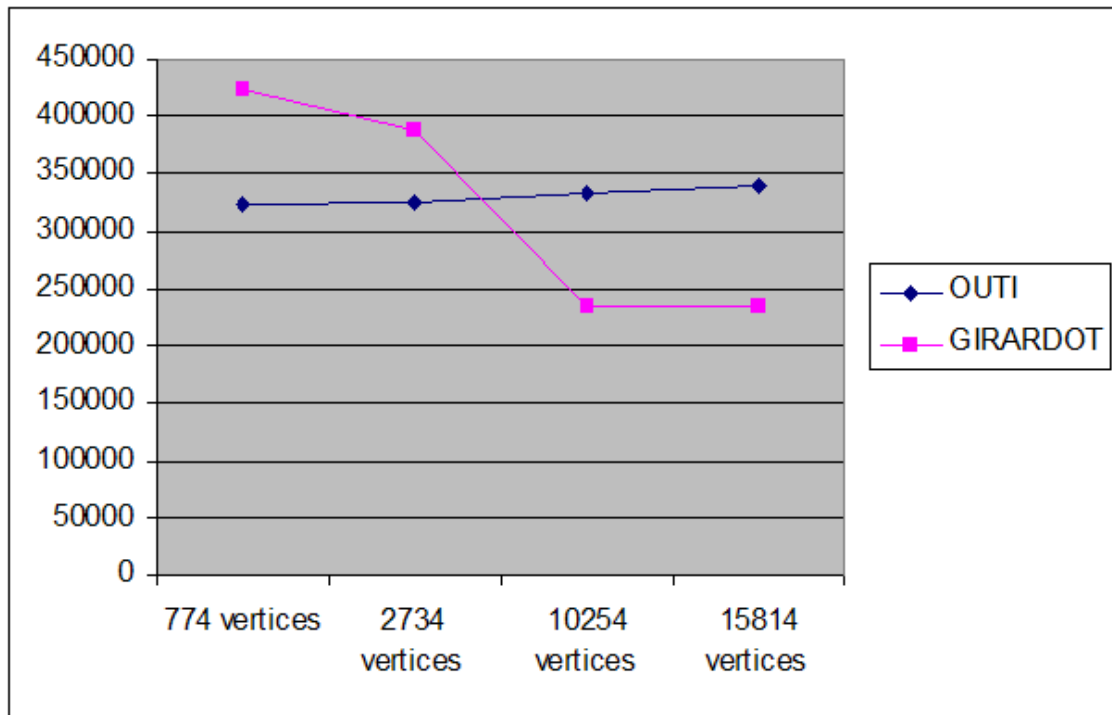


Fig. 5.5: Average time per vertex graph, changing model complexity.

From figure 5.5, it is made clear that model complexity is not the determining factor for single vertex processing time. The hardware used certainly plays a very important role, for Girardot seems to be much faster when more triangles are processed, while Outi isn't. However, as it is expected, the total calculation time is greatly affected by the amount of vertices present in the scene, for this time is multiplied by the total amount of vertices to obtain the total processing time.

⁴ Full data sets of the benchmarks are available in files `benchmark-outi.dat` and `benchmark-girardot.dat`, included in the companion CD.

5.8.2 Average time per vertex, changing hemicube size

This test is used to determine how the hemicube size affects the total computing time, and is deduced entirely from test 2.

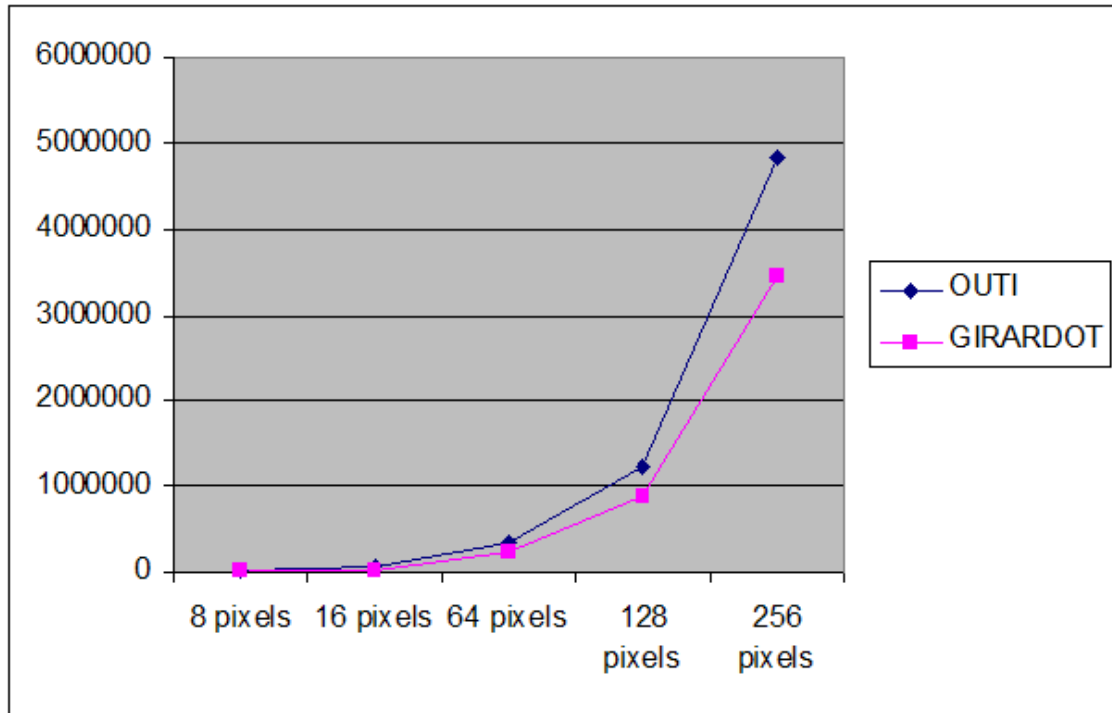


Fig. 5.6: Average time per vertex graph, changing hemicube size.

Figure 5.6 indicates that the actual determining factor for single processing time is the hemicube size. As the hemicube is enlarged, the amount of pixels to include in the calculations grows in a squared proportion. This is very important, for each time the hemicube is enlarged, the total calculation time grows by almost a whole order of magnitude. Hardware is certain to affect this, but not in a dramatic way. Since hemicube size is directly related to the precision of the calculations, it's very important for applications to tune this very carefully.

5.8.3 GPU/CPU workload ratio, changing hemicube size

Since hemicube size proved to be much more decisive than model complexity in total computing time, it is then necessary to determine how well the GPU/CPU division in the shader and host programs respectively is performed. In other words, if this whole development is worth the effort.

Figure 5.7 shows that for all scenarios, even though close to equal workload, more time is devoted to applying the BRDF and decoding the HDR data, which is performed at the CPU, than to actually rendering the scene, which is performed at the GPU; but it's also seen that as the hemicube is enlarged, the post-processing effort is increased. This division is apparently independent to the hardware used, but it is most possible that this can be used to measure how

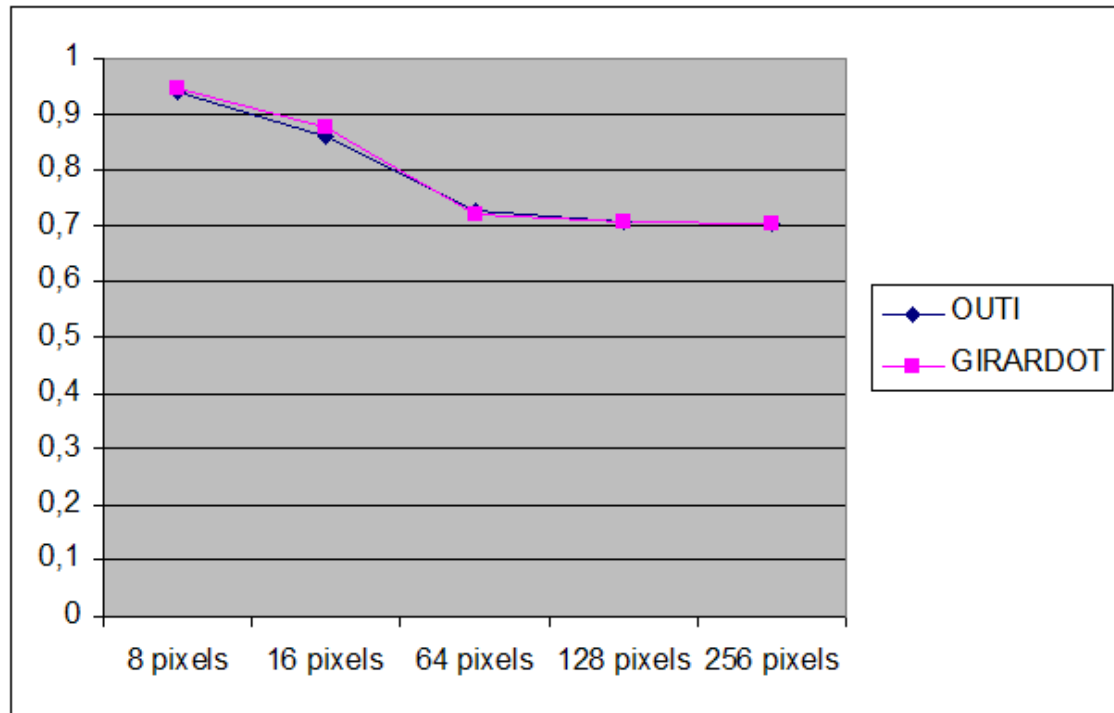


Fig. 5.7: GPU/CPU workload ratio graph, changing hemicube size.

the CPU and GPU in a single computer are related.

6. CONCLUSIONS AND AFTERTHOUGHTS

6.1 *Results*

Looking back on the developed work, it can be concluded that developing a global illumination renderer takes much more than just knowing the theory. There are many problems during development that aren't addressed by the theory, and must be considered individually. Some of the solutions presented in this work worked pretty good, such as HDR, but many others didn't. The main objective of this work, which was to motivate developers into shaders and global illumination, as well as to illustrate the technical difficulties experienced by such a developer, was heavily considered and evaluated.

6.2 *Motivation for future developments*

With this work, it is expected for developers and students to consider including shaders, as well as global illumination into their computer graphics studies. This work only covered a basic level of both technologies, and yet it required a large amount of work to get them working. Further development on either shaders or global illumination is likely to produce more spectacular results, but it's the inclusion of shaders into global illumination that is likely to yield new breakthroughs in computer graphics.

Future developments in this area require very little introductions to the topic. More complex algorithms such as Monte Carlo approximations to physical models require a very solid understanding of probability and stochastic methods for them to be included. But apart from that, only basic computer graphics and programming skills are required. It is this simplicity that should motivate further studies in these areas.

6.3 *Tool usage conclusions*

This work was developed with quite an unusual set of tools for academic works. Working with tools from the same vendor (in this case, Microsoft) for every aspect of the application is quite a pleasing experience, because everything is meant to work together, so no surprises are ever found. Microsoft tools in particular tend to give the developer great control over the process being worked on, as well as more than one way to perform the same action, in order to allow optimizations on different aspects of the program.

These tools worked very good during the development of this work, and future developers are encouraged to consider them within the tools for their own work.

6.4 Per-vertex illumination

Most radiosity renderers, as well as global illumination algorithms that must compute lighting on tessellated models, usually perform lighting calculations per-triangle instead of per-vertex. The reasons for this are not clear, and the demo program was developed so it would use per-vertex lighting calculations. This proved to be a very complicated problem, which required the use of the position displacement technique explained in section 4.5.3. Although this hack has little repercussions in the calculation's accuracy, it does create artifacts that affect the graphical quality of the result around the corners. Although in the real world there are no sharp corners, illumination is considered on an infinitely tessellated mesh, so this is not a problem.

A very important lesson learned with the development of this program is that future radiosity renderers should calculate lighting per-triangle and not per-vertex to avoid such artifacts.

6.5 CPU/GPU bottleneck

As it was clearly illustrated with the benchmarks, around half of the calculation time is executed in the CPU, while the other half is executed in the GPU. Moving more code to the shader should make execution faster, but it should be possible to tune the algorithm so half the processing time is performed in the CPU and the other half in the GPU. If this is achieved with synchronization techniques, it should be possible to have these two pieces of code to be executed in different threads, and since they are being executed in different processors, processing time could be cut up to half. While this problem is not as trivial as it sounds, cutting processing time to half should be a very interesting research field in the field of global illumination performance enhancements.

6.6 Classware

This work presented illumination from a theoretical and comprehensive point of view. The path followed by this work could be adapted for educational purposes, where theory is presented with the illumination models, and practice is achieved by the use of shaders.

6.7 Future work

6.7.1 More GPU work

As explained before, moving more calculations to the GPU would accelerate calculations. Remembering that programmable shaders can only solve those problems found inside the rendering pipeline, at first sight it's clear that it's possible to move the multiplier map usage and material application to a post-processing shader. In addition to that, it is also possible to perform the irradiance sum inside another post-processing shader that renders the current hemicube into a smaller box, with antialiasing activated, to minimize CPU workload. These are some, but certainly not all of the possible additions for accelerating the application.

6.7.2 *Post-processing effect shaders*

Shaders have great usage in post-processing effects for realistic image rendering. Composing various shaders that apply techniques such as HDR glare or lens flares fit properly into a radiosity renderer.

6.7.3 *Complex BRDFs*

This work focused solely on diffuse surfaces. The multiplier maps can be modified to fit new BRDFs, including those that depend on the viewer's point of view -even if a static images are rendered- to simulate thousands of different materials. There is currently a large amount of effort in BRDF development, which can be applied into global illumination renderers.

6.7.4 *Using geometry shaders*

As of the time this work is being written, geometry shaders are still in beta stage. They weren't researched on this work, but if they make it possible to define the tessellation for different parts of the mesh, it should be possible to dynamically alter the tessellation of the mesh to increase the level of detail where it's needed, thus optimizing the amount of calculations required, perhaps to implement real-time radiosity in the near future.

6.7.5 *Other global illumination algorithms*

Radiosity is by no means the only global illumination algorithm available. There are many more modern algorithms such as photon mapping, which claim to achieve better graphical quality, some with faster rendering times. Source [3], which was widely used for the development of this work, contains a comprehensive review of the most useful global illumination algorithms, all of which should be investigated for possible GPU optimizations.

APPENDIX

A. DEMO PROGRAM BASE CODE

The program was based on the “EmptyProject” sample included with DirectX 9.0 SDK (February 2006), chosen because it has GUI elements already included, and all of the framework is already initialized for proper Direct3D usage. After the sample was chosen, every “EmptyProject” string was replaced to “Radiosity”. The class name was renamed to `Radiosity`, a new file called `Radiosity_Render.cs` was created, and the `Radiosity` class was converted to a `partial` class. The new file was then meant to hold variables and methods related to the actual rendering process, while the old `Radiosity.cs` was meant to hold initialization details. Most of the variables at this point, and throughout the development of the program weren’t modified:

- `Framework radiosityFramework`: The Direct3D framework. This class is provided by the DXMUT framework, and was not modified in any way during the development of the demo program.
- `Font statsFont`: The current rendering font.
- `Sprite textSprite`: A sprite to hold the rendered text.
- `ModelViewerCamera camera`: The camera. This variable is very important, for it intercepts mouse commands in order to properly set its member matrices, which will be then passed to the shaders.
- `bool isHelpShowing`: Defines whether or not the help text is being displayed.
- `Dialog hud`: A container dialog for the standard controls (displayed at the top-right corner of the window).
- `Dialog controlUi`: A container dialog for custom controls (displayed at the bottom-right corner of the window). Further controls were later added to this dialog.

When the need to expand the UI was seen, another file that defined the same partial class, called `Radiosity_UI.cs` was also created. This file would hold all variables and methods related to UI creation and events. It contained the following methods:

- `void RenderText()`: Called each frame to create the status and help texts. All of the control code was maintained, only changing the actual text contents and rendering colors.
- `void OnKeyEvent()`: Handles the F1 key press to toggle the help display.

- `void InitializeApplication()`: This method originally created UI elements. This code was then moved to a more appropriately named `void InitializeUI()` method, and the contents of this method were changed only to call `InitializeUI()`. Other initializations, such as input or audio device creation should be inserted in this method.
- `void InitializeUI()`: This method Creates UI elements on their respective locations. This method was initially stripped of all of its non-standard controls, and as new options were added, the contents of this method were modified.

The following UI events were left, others were removed, along with their creation code in `void InitializeUI()`:

- `void OnChangeDeviceClicked()`: Called when the “Change Device” button is clicked. This method calls a series of functions inside the DXMUT framework and creates a UI for changing device settings as well as rendering settings.
- `void OnFullscreenClicked()`: Called when the “Toggle full screen” button is clicked. This method calls the necessary functions to set and unset fullscreen mode automatically within the DXMUT framework.
- `void OnRefClicked()`: Called when the “Toggle reference” button is clicked. This method switches between the selected device and the reference rasterizer, a software-only device that implements the entire DirectX function set, and can be used to verify compatibility issues at the expense of a severe performance penalty.

The following methods were left in `radiosity.cs` and control the creation of the DirectX environment, and were pretty much left unchanged:

- `bool IsDeviceAcceptable()`: Called by the DXMUT framework whenever a new device is set. New devices must be checked for compatibility with the required functionality, and can be accepted or rejected. The method originally required a back buffer that supports alpha blending, and it was left like that. The final application doesn't require alpha blending, so this can be safely removed.
- `bool ModifyDeviceSettings()`: Called by the DXMUT framework immediately before a new device is created, so the application may choose to change the device settings for additional requirements. This function was slightly modified to allow shader debugging, as well as to change the back buffer format for encoding HDR data. Refer to section 5.7 for more information regarding this.
- `void OnCreateDevice()`: Called by the DXMUT framework immediately after the device is created. Objects dependent on the device are created in this method, as well as effect objects. Objects created in here using `Pool.Default` are not automatically released by the garbage collector, so they *must* be explicitly released in `OnDestroyDevice()`. No objects of this nature were created in this method, however. This method also sets the camera parameters, which were modified to be positioned at a distance of 600 units in the Z^+ axis.

- `void OnResetDevice()`: Called by the DXMUT framework immediately after the device has been reset, such as when the window is resized, or moved outside of the desktop space. Creation of device-dependent objects, such as models, take place in this method. Objects created in here using `Pool.Default` are not automatically released by the garbage collector, so they *must* be explicitly released in `OnLostDevice()`.
- `void OnLostDevice()`: Called by the DXMUT framework when the Direct3D device has been lost, and before the device is reset. This method takes care of explicit disposal of device dependent objects, created on `OnResetDevice()`.
- `void OnDestroyDevice()`: Called by the DXMUT framework when the Direct3D device is destroyed, which happens when the device is going to be replaced with a new device (such as when the application is switching between windowed and full screen modes, or when the application is terminated). Global device dependent objects, created on `OnCreateDevice()` are disposed of in this method. This program, however, doesn't define any objects of this nature¹.
- `IntPtr OnMsgProc()`: This method processes window messages, and is similar to the Win32 `MessageProc()` function. The default behavior of this implementation consists of passing the message, in order to: the user controls, the default controls, and finally to the camera. The first one to handle the message will prevent the rest to do so. If the message is not handled, it is passed back to the DXMUT framework, so it can handle the message. This behavior was not changed in the demo program.
- `static int Main()`: This is the program entry point. The window is created in this function, and finally `radiosityFramework.MainLoop()` is called to run the application.

As mentioned before, `Radiosity_Render.cs` was created to contain all rendering code. The following methods are located in it:

- `void OnFrameMove()`: Called by the DXMUT framework once each frame, before it's rendered. This method should contain updates to the scene, such as camera updates, but not rendering code, which goes in `void OnFrameRender()` instead.
- `void OnFrameRender()`: Called by the DXMUT framework once each frame, and rendering code is called in this method. This method suffered many changes as rendering became complex. The current behavior of this method is to clear the Z buffer and the target to a specific background color, which was chosen to be a dark green of value `0x8C003F3F`.

With this base code, the program creates a window, sets it up to contain a Direct3D device, creates a device, resets it when necessary; and for each frame, clears the window to a dark green background. This is the initial behavior from which the rest of the application was built onto.

¹ Models however, could be created along with the device, and be disposed of inside `void OnDestroyDevice()`. Changing this behavior will only reload models when the device is destroyed, which can be a good thing or a bad thing, depending on the desired behavior. The demo program, however, reloads models whenever the device is reset.

B. DEMO PROGRAM MESH LOADING CODE

In order to choose the appropriate model set to load, a combo box was created so the user could choose the desired model, whose output value is stored in variable `model`. The process of loading a model consists of the following steps:

1. Clear the model list `meshList` with the following code:

```
for (int i = 0; i < meshList.Count; i++)
{
    meshList[i].mesh.Dispose();
}
meshList.Clear();
```

2. Choose the appropriate model set based on the current value of variable `model`. This is performed with a `switch` statement, where a single `case` label contains code similar to the following one:

```
switch (model)
{
    // ...
    case 0x0005:
        AddMeshFromFile("Models\\Sphere.x");
        AddMeshFromFile("Models\\Torus.x");
        AddMeshFromFile("Models\\box-cornell-tess20.x");
        vecLightDirection =
            new Vector4(0.0f, -1.0f, 0.0f, 0.0f);
        break;
    // ...
}
```

It's also necessary to choose the light direction for the local illumination algorithms. This is stored in variable `vecLightDirection`, and each model defines a specific light location ¹.

Method `AddModelFromFile()` loads a model from a file and appends it to `meshList` in the following manner:

¹ Three dimensional vectors in space are usually stored as *XYZ* values, but it's more natural memory-wise to store them in `Vector4` structures with the last parameter *W* discarded.

- (a) Create a new `CompositeMesh` object for later insertion in the list:

```
CompositeMesh newMesh = new CompositeMesh();
```

- (b) Call `Mesh.FromFile` to load the mesh into `Pool.Default`. This also loads the material information inside array `materials`:

```
newMesh.mesh=Mesh.FromFile(
    Utility.FindMediaFile(path),
    MeshFlags.VbManaged,
    device,
    out materials);
```

- (c) Clone the mesh to open room for an additional color component, which will be used for global illumination in section 5.6, and calculate normals if they don't exist yet:

```
CalculateNormals = (newMesh.mesh.VertexFormat &
    VertexFormats.Normal)==0 ? true : false;
Mesh TempMesh = newMesh.mesh.Clone(
    newMesh.mesh.Options.Value,
    VertexFormats.PositionNormal |VertexFormats.Diffuse,
    device);
newMesh.mesh.Dispose();
newMesh.mesh = TempMesh;
if (CalculateNormals)
    newMesh.mesh.ComputeNormals();
```

- (d) Make room to copy the vertices and copy the data inherent to the actual vertices in the vertex array:

```
newMesh.vertices =
    new PositionNormalDiffuseEmmissiveColorVertex
        [newMesh.mesh.NumberVertices];
GetVertexPosNormals(newMesh.mesh,
    ref newMesh.vertices);
```

`GetVertexPosNormals()` copies data from the vertex buffer into the vertex array, in file `Radiosity_Render_Radiosity.cs` in the following manner:

- i. Lock the entire extent of the vertex buffer and acquire the internal data pointer by casting it to a custom vertex type pointer that matches the vertex format used when the mesh was cloned:

```

GraphicsStream s =
    CurrentMesh.LockVertexBuffer(LockFlags.None);
PositionNormalDiffuseVertex* p;
p = (PositionNormalDiffuseVertex*)
    (s.InternalDataPointer);

```

- ii. For each vertex, copy its position, normal and diffuse color. The normal vector is then normalized:

```

for (int i = 0; i < CurrentMesh.NumberVertices; i++)
{
    vertexList[i].Position = p[i].Position;
    vertexList[i].Normal = p[i].Normal;
    vertexList[i].Normal.Normalize();
    vertexList[i].Color = p[i].Diffuse;
}

```

It's important to notice that the vertex buffer stores vertices with only Position, Normal and Diffuse components, while the vertex array contains additional Emmissive and Color components. When global illumination is calculated, the Color component will hold the current computed color for the vertex, while the Diffuse component will hold the material color. The Emmissive component will hold, as its name implies, the emmissive color for the vertex.

The Diffuse and Emmissive components of the vertex are not inherent to each vertex, but to the corresponding material, so they are filled in later.

- iii. After all vertices have been processed, unlock the vertex buffer:

```

CurrentMesh.UnlockVertexBuffer();

```

- (e) Copy the data inherent to each material into the vertex array:

- i. Iterate the material array, and for each material add the relevant information stored in `ExtendedMaterial.Material3D`:

```

for (int i = 0; i < materials.Length; i++)
{
    newMesh.materialList.Add
        (materials[i].Material3D);
    // ...
}

```

- ii. For each material, retrieve its triangle list and lock the index buffer:

```

IndexBuffer indBuffer;
int numStrips;
int numIndices;
GraphicsStream stripLengths;
indBuffer =
    Mesh.ConvertMeshSubsetToStrips(
        newMesh.mesh, i, MeshFlags.SystemMemory,
        out numIndices, out stripLengths,
        out numStrips);
short[] indices = (short[])
    (indBuffer.Lock(0, typeof(short),
        LockFlags.None, numIndices));

```

- iii. For each index in the index buffer, the correspondent vertex is filled with the Diffuse and Emmisive components of the material. Note that it's possible to use `Material.Diffuse` and `Material.Emmisive` instead of `Material.DiffuseColor` and `Material.EmmisiveColor`. However, both values are not equal, for the latter uses floats and not bytes as the former. This allows the use of a greater range of color components which will be used later in section 5.7 when HDR lighting is considered.

Because of this structure, vertices that are shared by more than one triangle are considered multiple times:

```

for (int j = 0; j < indices.Length; j++)
{
    newMesh.vertices[indices[j]].Diffuse =
        new Vector4(
            materials[i].Material3D.DiffuseColor.Red,
            materials[i].Material3D.DiffuseColor.Green,
            materials[i].Material3D.DiffuseColor.Blue,
            materials[i].Material3D.DiffuseColor.Alpha);
    newMesh.vertices[indices[j]].Emmisive =
        new Vector4(
            materials[i].Material3D.EmissiveColor.Red,
            materials[i].Material3D.EmissiveColor.Green,
            materials[i].Material3D.EmissiveColor.Blue,
            materials[i].Material3D.EmissiveColor.Alpha);
}

```

- iv. Then, unlock the index buffer:

```
indBuffer.Unlock();
```

3. Finally, after the mesh has been loaded, choose ambient and color light colors appropriate for the specific scene. In the demo program, all models will work with a dark grey ambient light and a light grey directional light. Colors are selected as `Vector4` structures in the following manner:

```
for (int i = 0; i < materials.Length; i++)
{
    newMesh.materialList.Add
        (materials[i].Material3D);
    // ...
}
```

Now that models are properly loaded and set up, they are ready to be rendered. The only part left to consider is the code to dispose of them. Since meshes are loaded into `Pool.Default`, they must be explicitly disposed of inside `OnLostDevice()`. Performing this is done in the same way the mesh list was disposed of, before the models were loaded:

```
for (int i = 0; i < meshList.Count; i++)
{
    meshList[i].mesh.Dispose();
}
meshList.Clear();
```

Disposing of mesh objects in two different places seems redundant when the device is reset, such as when the window is resized, for they are disposed of once when the device is reset, and again when it's created again; if the code that disposes of the meshes when the models are loaded is removed, the program crashes when the scene is changed, and then the window is resized, because the original mesh remains inside `Pool.Default` and is not disposed of when the device is reset as illustrated on figure B.1.

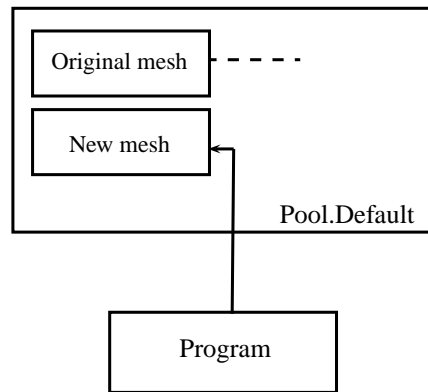


Fig. B.1: Pool.Default status when two meshes are loaded

C. COLOR PLATES



Fig. C.1: A simple scene, rendered using technique C. This technique runs in real time at more than 30 frames per second.



Fig. C.2: A simple scene, rendered using technique CA. Ambient color has obscured the entire image constantly. This technique runs in real time at more than 30 frames per second.

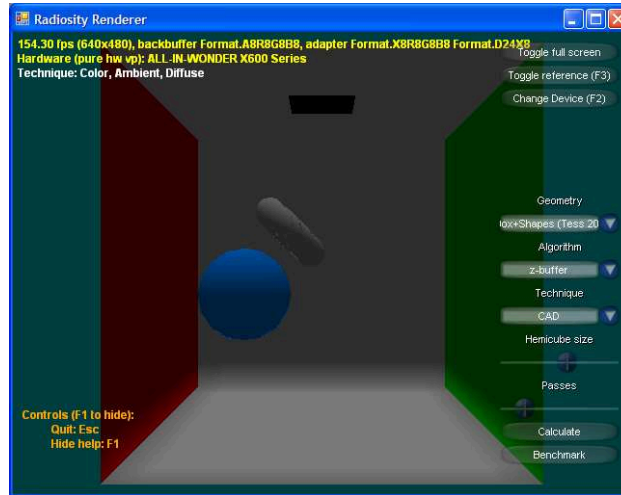


Fig. C.3: A simple scene, rendered using technique CAD. Shapes form simple lighting patterns. This technique runs in real time at more than 30 frames per second.



Fig. C.4: Detail of specular lighting, rendered using technique CAS. Note how per-vertex lighting forms artifacts in the floor. This technique runs in real time at more than 30 frames per second.

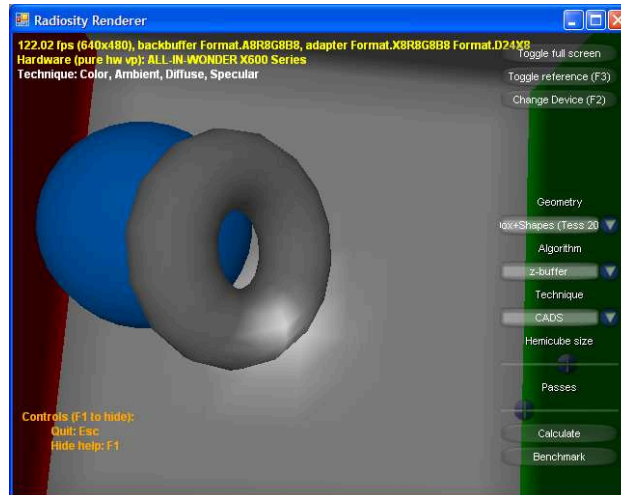


Fig. C.5: Detail of diffuse and specular lighting, rendered using technique CADs. This technique runs in real time at more than 30 frames per second.



Fig. C.6: Detail of specular lighting, rendered using technique CAS(P). Per-pixel lighting has solved all artifacts, and lighting is now smooth. This technique runs in real time at more than 30 frames per second.

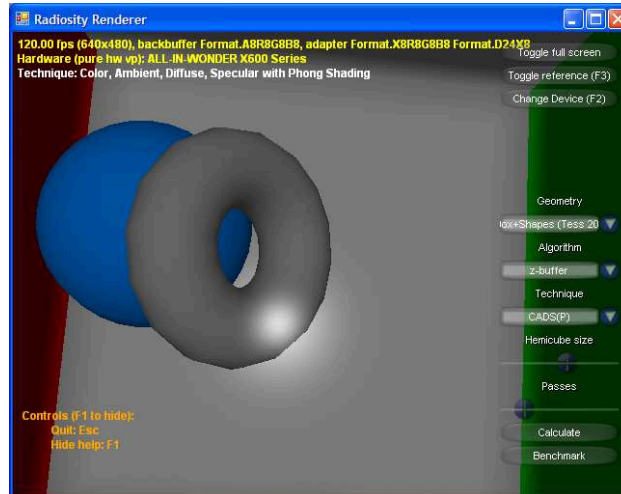


Fig. C.7: Detail of diffuse and specular lighting, technique CADSP. This technique runs in real time at more than 30 frames per second.

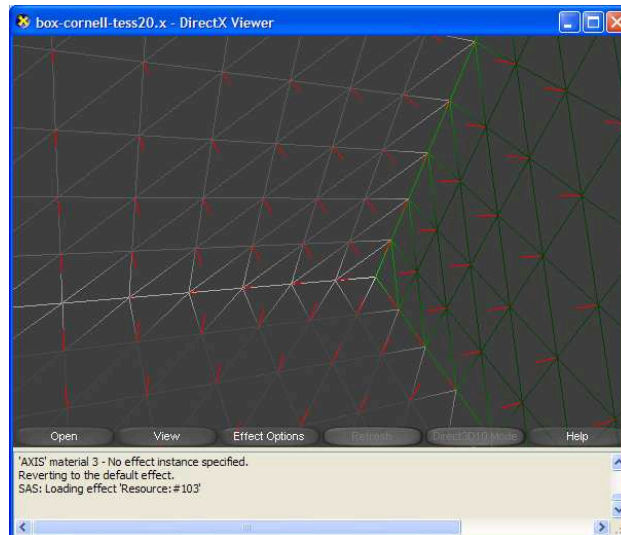


Fig. C.8: Close up of normals in the box model. Normal vectors are represented in red.



Fig. C.9: An empty box, similar to the Cornell box, rendered using the radiosity method described in this work. This scene took 58 seconds to calculate, but is presented in real time at more than 30 frames per second.

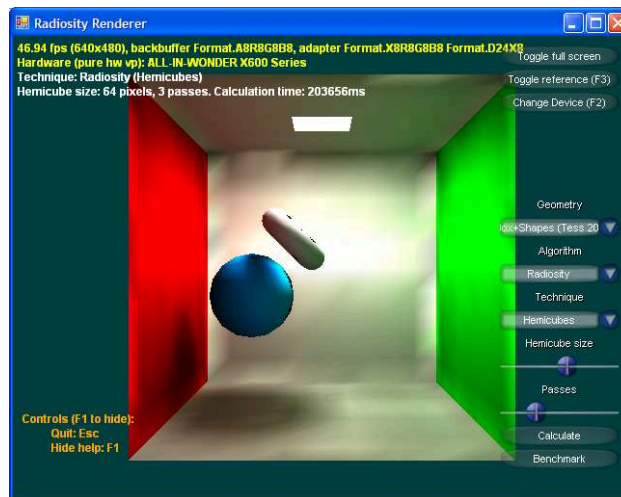


Fig. C.10: A box with two shapes, radiosity rendered with an HDR light source of color (100.4, 96.8, 90.0) (almost 100 times brighter than white). The sphere projects a soft shadow on the floor, and the torus above the sphere projects a soft shadow on it. This scene took 203 seconds to calculate, but is presented in real time at more than 30 frames per second.

BIBLIOGRAPHY

- [1] James F. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.
- [2] Michael F. Cohen and John R. Wallace. *Radiosity and realistic image synthesis*. Morgan Kaufman Publishers, Inc., 1993.
- [3] Phillip Dutré, Phillipe Bekaert, and Kavita Bata. *Advanced Global Illumination*. A K Peters Ltd., 2003.
- [4] Hugo Elias. Tgltlsbfssp: Radiosity. (<http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm>). site last visited on may 22, 2006.
- [5] Wolfgang Engel. *ShaderX²: Shader programming tips and tricks with DirectX 9.0*. Charles River Media, first edition, 2003.
- [6] Wolfgang Engel. *Programming Vertex and Pixel Shaders*. Charles River Media, first edition, 2004.
- [7] Wolfgang Engel. *ShaderX³: Advanced rendering with DirectX and OpenGL*. Charles River Media, first edition, 2005.
- [8] Wolfgang Engel. *ShaderX⁴: Advanced rendering techniques*. Charles River Media, first edition, 2006.
- [9] Randima Fernando. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Addison-Wesley, 2004.
- [10] J. H. Lambert. *Photometria sive de mensura de gratibus luminis colorum umbrae*. 1760.
- [11] F. E. Nicodemus. Reflectance nomenclature and directional reflectance and emissivity. *Appl. Opt.*, 9:474–1475, 1970.
- [12] F. E. Nicodemus, J. C. Richmond, J. J. Hsia, I. W. Ginsberg, and T. Limperis. Geometric considerations and nomenclature for reflectance. Monograph 161, National Bureau of Standards (US), oct 1977.
- [13] Adrian Perez and Dan Royer. *Advanced 3-D Game programming using DirectX 7.0*. Wordware Publishing Inc., 2000.

-
- [14] Bui Tuong Phong. *Illumination for computer generated images*. PhD thesis, University of Utah, 1973.
 - [15] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.
 - [16] Sebastien St-Laurent. *Shaders for game programmers and artists*. Thomson Course Technology, first edition, 2004.
 - [17] Greg Ward. *Real pixels*, pages 80–84. Morgan Kaufmann, 1994.