1. Abstract

Global illumination is a very complex area of rendering. Traditional attempts have been made with *offline* renderers: renderers that do not run at real-time speeds and are tailored to produce photorealistic images. However, these renderers can take from hours to days to process a scene. The ultimate challenge for any graphics programmer, therefore, is to write a global illumination renderer that runs in *real-time*. This project attempts to do just that.

The first stage of this project involved investigating techniques that had been previously developed or implemented. These methods were documented, compared and contrasted. Their major successes and failures were also analysed, stating why they were, or why they were not, good selections for a real-time global illumination solution. This phase involved a significant amount of research and would eventually lead to the development of an algorithm that I have named *Photon Splatting*.

The second stage of this project was the implementation. The most viable algorithm from the first stage was selected for further research and development. In this case *Irradiance Cache Splatting* was chosen. This algorithm, however, was not entirely suited for the speed requirements given by the specification, so it was adapted and optimised to create *Photon Splatting*. This algorithm was then incorporated into a real-time, interactive engine that I developed using Microsoft's DirectX.

This project has demonstrated that global illumination *is* possible on a graphics processor and it is capable of rendering a simple scene at interactive speeds. It is also able to produce the same quality output as a low resolution photon map. However, the project also shows that perhaps graphics hardware is not quite ready for global illumination just yet: while the algorithm is able to render a simple scene well, it would suffer extraordinarily when presented with a scene that is more complex.

2. Acknowledgements

2.1 Dedication

I would like to dedicate this project to Marjorie Watson, my Gran, who passed away in February 2006. I feel that the only way I can thank her for all she has done for me is by working as hard as I can. A bit of "sweat and elbow-grease", as she would have said, which this project has certainly involved plenty of.

2.2 Thanks To…

I would like to thank the following people:

Tyrone Davison,
for his merciless scrutiny and
expert opinion on the topic.

Christopher Agnew,
for telling me that 5,000 words is
easy, man, seriously…

Alice "Miss Wiggle" Darcy,
for proof-reading, cooking
and forcing me to take a break
now and again,

and my other proof readers,
Ali Parr,
Dani Wright
and **Luke Nockles**
for giving up their precious time.

3. Contents

3.1 Main Document

1. Abstract.....i

2. Acknowledgements.....ii

2.1 Dedication.....ii

2.2 Thanks To.....ii

3. Contents.....iii

3.1 Main Document.....iii

3.2 Appendices.....vi

4. Introduction.....10

5. Research Methods.....12

5.1 Finding Information.....12

5.2 Using the Research.....12

6. Current Solutions.....14

6.1 Light-Mapping.....14

6.2 Pre-Computed Radiance Transfer & Spherical Harmonics.....15

6.3 (Ir)Radiance Cache-Based Methods.....16

6.3.1 Radiance Mapping.....16

6.3.2 Instant Radiosity.....17

6.3.3 Irradiance Cache Splatting.....18

7. Photon Splatting.....20

7.1 Photon Mapping Overview.....20

7.1.1 Introduction.....20

7.1.2 Generating the Light Data.....20

7.1.3 Finding the Illumination.....20

7.2 Photon Splatting.....21

7.2.1 What is Photon Splatting?.....21

7.2.2 Photon Splatting versus Irradiance Cache Splatting.....21

7.2.3 The Faster Model.....22

8. From Paper to Pixels 25

8.1 Approaching Design.....25

8.1.1 Middleware?.....25
8.1.2 Setting Goals.....25
8.1.3 Designing the Renderer.....26
8.2 Implementing Photon Splatting 28
8.2.1 How Analysis and Design Helps.....28
8.2.2 Building the Implementation.....29
8.2.3 Writing the Algorithm.....30
8.2.4 Correcting Design Flaws.....31
8.2.5 Learning the DirectX API.....31
8.2.6 An Example Problem.....32
8.2.7 Mesh Tessellation.....33

9. Evaluation.....34
9.1 Successes.....34
9.1.1 Research Strengths.....34
9.1.2 Design Strengths.....34
9.1.3 Implementation Strengths.....35
9.2 Failures.....36
9.2.1 Research Weaknesses.....36
9.2.2 Design Weaknesses.....37
9.2.3 Implementation Weaknesses.....38
9.3 Improvements.....39
9.3.1 Variably Size Splats.....39
9.3.2 Floating-Point Buffers.....40
9.3.3 Changing the Photon Sampling Method.....40
9.4 Optimisations.....40
9.4.1 Performance Hits.....41
9.4.2 Selective Photon Splatting & Combining with Other Models.....41
9.4.3 Using a Lower Resolution Splat Buffer.....41
9.4.4 Frustum Culling.....42
9.4.5 Space Partitioning.....42
9.5 Knowledge Gained.....43

10. Conclusions.....45

11. References.....47

12. Appendix A: Project Specification.....50

12.1 Motivation & Aims.....50
12.2 Areas of Research.....50
12.3 Schedule.....51
12.4 Criteria.....51

13. Appendix B: Current Solutions.....53
13.1 Light Mapping.....53
13.1.1 Light-Mapping Overview.....53
13.1.2 Finding the Illumination Contribution.....55
13.1.3 How are Light-Maps Used in Rendering?.....56
13.1.4 Storing Light-maps.....57
13.1.5 How Light-Mapping is Useful.....57
13.1.6 How Light-Mapping is Flawed.....58
13.2 Pre-computed Radiance Transfer (PRT).....59
13.2.1 What is Pre-computed Radiance Transfer?.....59
13.2.2 How Does Pre-computed Radiance Transfer Work?.....59
13.2.3 Using the Pre-computed Data.....62
13.2.4 How Pre-computed Radiance Transfer is Useful.....63
13.2.5 How Pre-computed Radiance Transfer is Flawed.....63
13.2.6 Associated Legendre Polynomials.....64
13.3 (Ir)Radiance-Based Methods.....65
13.3.1 What is (Ir)Radiance Caching?.....65
13.3.2 Traditional Radiance Cache.....66
13.3.3 Radiance Mapping.....67
13.3.4 Instant Radiosity.....68
13.3.5 Irradiance Cache Splatting.....68
13.3.6 How (Ir)Radiance Caching is Flawed.....71

14. Appendix C: Photon Splatting.....72
14.1 Introduction to Photon Splatting.....72
14.2 Photon Computation.....72
14.3 The Splatting Algorithm.....73
14.4 The Implementation.....74
14.4.1 Implementation Overview.....74
14.4.2 Speed and Efficiency.....75
14.5 Results and Conclusions.....76

15. Appendix D: Image Output.....78

15.1 Sphere Demonstration.....78

15.2 Cornell Box Scene.....80

15.3 Off-Screen Passes.....81

15.4 The Splat Buffer.....82

15.5 The Scene without Diffuse Inter-reflections.....83

16. Appendix E: Shader Listings.....84

16.1 The Splatting Algorithm.....84

16.2 Splat Buffer Combiner.....87

17. Appendix F: Document Statistics.....89

3.2 Appendices

1. Abstract.....i

2. Acknowledgements.....ii

2.1 Dedication.....ii

2.2 Thanks To.....ii

3. Contents.....iii

3.1 Main Document.....iii

3.2 Appendices.....vi

4. Introduction.....10

5. Research Methods.....12

5.1 Finding Information.....12

5.2 Using the Research.....12

6. Current Solutions.....14

6.1 Light-Mapping.....14

6.2 Pre-Computed Radiance Transfer & Spherical Harmonics.....15

6.3 (Ir)Radiance Cache-Based Methods.....16

6.3.1 Radiance Mapping.....16

6.3.2 Instant Radiosity.....17

6.3.3 Irradiance Cache Splatting.....18

7. Photon Splatting.....20

7.1 Photon Mapping Overview.....20

7.1.1 Introduction.....20

7.1.2 Generating the Light Data....20
7.1.3 Finding the Illumination....20
7.2 Photon Splatting....21
7.2.1 What is Photon Splatting?....21
7.2.2 Photon Splatting versus Irradiance Cache Splatting....21
7.2.3 The Faster Model....22

8. From Paper to Pixels 25
8.1 Approaching Design....25
8.1.1 Middleware?....25
8.1.2 Setting Goals....25
8.1.3 Designing the Renderer....26
8.2 Implementing Photon Splatting 28
8.2.1 How Analysis and Design Helps....28
8.2.2 Building the Implementation....29
8.2.3 Writing the Algorithm....30
8.2.4 Correcting Design Flaws....31
8.2.5 Learning the DirectX API....31
8.2.6 An Example Problem....32
8.2.7 Mesh Tessellation....33

9. Evaluation....34
9.1 Successes....34
9.1.1 Research Strengths....34
9.1.2 Design Strengths....34
9.1.3 Implementation Strengths....35
9.2 Failures....36
9.2.1 Research Weaknesses....36
9.2.2 Design Weaknesses....37
9.2.3 Implementation Weaknesses....38
9.3 Improvements....39
9.3.1 Variably Size Splats....39
9.3.2 Floating-Point Buffers....40
9.3.3 Changing the Photon Sampling Method....40
9.4 Optimisations....40
9.4.1 Performance Hits....41
9.4.2 Selective Photon Splatting & Combining with Other Models....41

9.4.3 Using a Lower Resolution Splat Buffer.....41
9.4.4 Frustum Culling.....42
9.4.5 Space Partitioning.....42
9.5 Knowledge Gained.....43

10. Conclusions.....45

11. References.....47

12. Appendix A: Project Specification.....50
12.1 Motivation & Aims.....50
12.2 Areas of Research.....50
12.3 Schedule.....51
12.4 Criteria.....51

13. Appendix B: Current Solutions.....53
13.1 Light Mapping.....53
13.1.1 Light-Mapping Overview.....53
13.1.2 Finding the Illumination Contribution.....55
13.1.3 How are Light-Maps Used in Rendering?.....56
13.1.4 Storing Light-maps.....57
13.1.5 How Light-Mapping is Useful.....57
13.1.6 How Light-Mapping is Flawed.....58
13.2 Pre-computed Radiance Transfer (PRT).....59
13.2.1 What is Pre-computed Radiance Transfer?.....59
13.2.2 How Does Pre-computed Radiance Transfer Work?.....59
13.2.3 Using the Pre-computed Data.....62
13.2.4 How Pre-computed Radiance Transfer is Useful.....63
13.2.5 How Pre-computed Radiance Transfer is Flawed.....63
13.2.6 Associated Legendre Polynomials.....64
13.3 (Ir)Radiance-Based Methods.....65
13.3.1 What is (Ir)Radiance Caching?.....65
13.3.2 Traditional Radiance Cache.....66
13.3.3 Radiance Mapping.....67
13.3.4 Instant Radiosity.....68
13.3.5 Irradiance Cache Splatting.....68
13.3.6 How (Ir)Radiance Caching is Flawed.....71

14. Appendix C: Photon Splatting.....72

14.1 Introduction to Photon Splatting.....72
14.2 Photon Computation.....72
14.3 The Splatting Algorithm.....73
14.4 The Implementation.....74
14.4.1 Implementation Overview.....74
14.4.2 Speed and Efficiency.....75
14.5 Results and Conclusions.....76

15. Appendix D: Image Output.....78
15.1 Sphere Demonstration.....78
15.2 Cornell Box Scene.....80
15.3 Off-Screen Passes.....81
15.4 The Splat Buffer.....82
15.5 The Scene without Diffuse Inter-reflections.....83

16. Appendix E: Shader Listings.....84
16.1 The Splatting Algorithm.....84
16.2 Splat Buffer Combiner.....87

17. Appendix F: Document Statistics.....89

4. Introduction

Global illumination (sometimes referred to as GI) is a term used by graphical scientists to describe the way light bounces between objects. It is an approximation of the *Light Transport Equation* (LTE) which is meant to cover every kind of light phenomena for a computer simulation.

This project was born from my desire to do something challenging and something that had not been explored much in the past. The dream of every programmer in the world of computer graphics is to create not only realistic output, but to also expand the field into new areas and introduce new concepts. With this project I wished to do just that: choose a particular aspect of global illumination that had not been explored previously and attempt to implement it on graphics hardware.

Generally speaking, global illumination is complex and much too computationally intensive for real-time applications, but some research has been done into approximating GI phenomena in real-time on a *Graphics Processing Unit* (GPU). Reflection and refraction are relatively simple phenomena and have been approximated in real-time with pleasing results. However, subsurface scattering and diffuse inter-reflection are the most complex elements of global illumination and only limited research has taken place in simulating them in at interactive speeds.

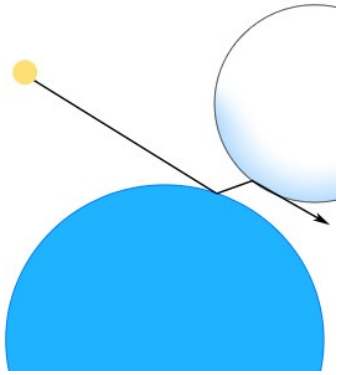


Figure 1 - Bouncing Light Ray

The research that I performed for this dissertation was centred on simulating *diffuse inter-reflection* in real-time. Diffuse inter-reflection is the term to describe the multiple bounces of a light ray before it reaches the eye as shown in Figure 1.

It is also referred to as *colour bleeding* and also – often incorrectly – as *radiosity*. Diffuse inter-reflection can be seen by taking a brightly coloured object and placing it near a white sheet of paper.

This project's intention was to research current implementations of diffuse inter-reflection and to then to implement a chosen method. This could be one that I developed or simply implementing a method that already existed, improving or approximating it if necessary. One of the primary goals was to cover at least *first bounce* diffuse inter-reflection in the implementation. This means that by the end of the project period, a working and interactive simulation of first bounce global illumination was achieved. This document is divided into sections that will pertain to an appropriate stage of research and development.

Research Methods explains my approach to researching material. Here I will cover where I searched for sources of relevant information, how I found these sources and how I was able to determine what information was useful. *Current Solutions* is an overview of the currently existing techniques that I decided to investigate, including an analysis of the research I performed and any conclusions that I drew. *Photon Splatting* details which algorithm from those that I investigated I have chosen to implement; it covers my analysis of the project and my justification for choosing it.

From Paper to Pixels regards the process of implementing the algorithm on the GPU in C++. It is composed of two major sections: the design phase is covered in *Approaching Design* and the implementation phase is covered in *Implementing Photon Splatting*. It describes how these phases were accomplished, including any problems that may have hindered the project. It also describes how the design phase helped complete the implementation phase.

Finally, there is the *Evaluation* and *Conclusion*. The former is a comparison of how well the algorithm worked with how it was *expected* to work. The latter is a summary of the observations made during the any aspect of the project and as a whole.

5. Research Methods

5.1 Finding Information

Global illumination is not a recent concept, yet it is only recently that any attempts have been made to extend any solutions into real-time. This has the unfortunate side effect of making research papers and books very difficult to find, so most of the research for this project was done on the Internet.

The primary method of finding sources was via Google.com. Most of the sources referenced here were white papers written in early 2005, which meant that they were relatively new and obscure. Other references included personal websites that featured tutorials or information on the subject matter that was being searched for. It is possible that the information given on websites was inaccurate, so research papers were references in preference over websites. However, websites do provide very up to date content, where as research papers are rarely modified after their submission.

All of the research that I performed was saved and stored on to my computer (if it was a digital reference) and occasionally printed for a hard-copy, or if the reference was a physical medium it was photocopied. Being careful not to breach copyright laws, this information was only selectively reproduced. Also, any notes that were taken were kept in an A4-sized blue book (referred to as the project book) alongside the date they were written on. This book allowed me to keep all of my important information in one place: it also stored ideas, "brainstorming" notes and a summary of each project meeting.

5.2 Using the Research

The information compiled from each of the sources was used in one of two situations. The former situation was researching current techniques (see Appendix B) where the information was collated and then summarised in the research reports. The latter situation was the development of the final algorithm. The algorithm was developed by mixing several techniques, so information was compiled together from various sources, selectively processed and combined to form a complete solution.

Extracting information from references involved reading the material, taking notes or highlighting certain sections and ensuring that I was able to understand it. These notes were then recorded in the appropriate section of the project book, for example any notes on Pre-computed Radiance Transfer were stored under that heading. Any notes that were either applicable to many headings or to none were kept at the front of the book for easy reference. It was thus that I was able to organise and efficiently retrieve information when writing any documents.

This project book co-existed with the provided project diary sheets. Each reference that I discovered was noted in full in the project book under the relevant week of discovery. The reference was then noted in the diary sheet for that week under its title (as opposed to the URL of the resource to prevent confusion of many links). This also meant that any references that I discovered were marked next to the relevant information that I extracted from it, instead of having a reference written on a diary sheet and not knowing what information it provided. I felt this to be a much tidier system.

6. Current Solutions

Current techniques were looked at for either inspiration for further development, or for the development of a new solution. In total, three categories of solution were looked at and documented appropriately. The results of this research can be seen in Appendix B. This section outlines how the research was conducted and how it helped me to devise my own solution.

6.1 Light-Mapping

The first solution looked at was light-mapping. As explained in Appendix B section 1, light-mapping is a method of storing pre-computed light data for each face of the geometry. It is an old technique that has been used in games since 1995 and it is only recently that games have moved to dynamic forms of lighting.

As the previous paragraph implies, the storage of pre-computed data means that light-mapping is not a dynamic solution[1]. While light-mapping may be flexible in what phenomena it can support, any changes to the geometry will immediately invalidate the light-map data. This means that animation of any kind is completely out of the question and made light-mapping a particularly bad choice as a basis for further study. However, light-mapping does allow some very sophisticated extensions such as multiple-bounce lighting calculations and caustics, but this can only be done in the pre-processing stage.

Another reason that light-mapping was not selected was that it is an aged. It has been central to providing lighting in many real-time applications since its inception and therefore has been thoroughly researched by others already. Further study of this technique would likely yield results that have already been discovered before.

Finally, light-mapping only classifies as a real-time global illumination technique by a technicality. In reality, it is only a method of storing lighting data and is not a method of calculating global illumination in real-time. However, this could also be said about Pre-computed Radiance Transfer, which is covered next.

[1] In this context, a dynamic solution would mean any kind of geometry deformation, such as animation.

6.2 Pre-Computed Radiance Transfer & Spherical Harmonics

Pre-computed Radiance Transfer (PRT) is another method of storing global illumination data for geometry. This immediately would suggest that it has the same flaws as light-mapping, but this is not the case. As opposed to simply storing what incoming light reaches each point, the pre-computed data in PRT is stored in what are known as *transfer vectors* and these contain information about how incoming light is *transmitted* by the medium. This means that we are not given data that tells us what light is hitting a point, but instead we are given data that tells us how to reflect, refract or otherwise change incoming light that reaches that point.

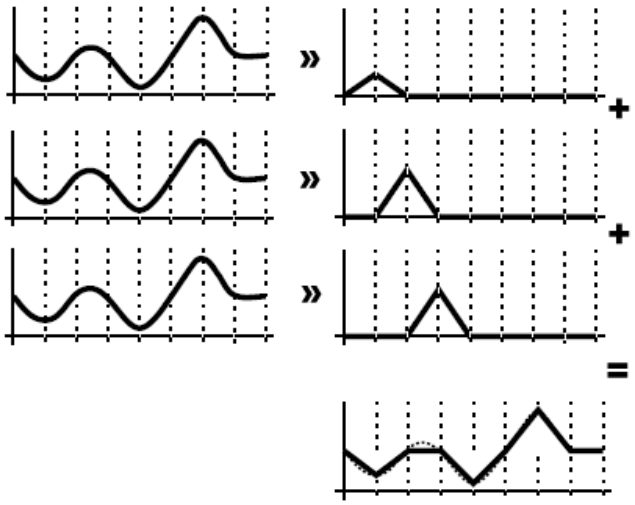


Figure 2 - Reconstructing a Light Signal

PRT involves the calculation of outgoing radiance (known as the *exit radiance* or *irradiance*) by reconstructing an analogue signal, just like real light. This is done using what is known as *spherical harmonic,* as shown in Figure 2.

Spherical harmonics is a subject in mathematics that extends from many theories which made it a particularly difficult subject to research. When documenting this subject it was almost impossible to determine what information should be omitted or included and a week had to be added to this subject's research phase in order to produce the research document in section 2 of Appendix B. Less than half of the information that was researched was included in the final write-up.

PRT is also flawed in certain ways. While it has the advantage of capturing many forms of phenomena dynamically, it is unable to transmit light to other objects. Also, it is only able to handle *affine transformations*[2] so the pre-computed data is only valid while the geometry remains the same. Just like light-mapping, this makes animation impossible. Finally, PRT can only handle low frequency (or "soft") lights because of the way the light signal is decompiled and reconstructed.

While PRT is interesting, vast and challenging, I felt that this method did not warrant further study. Though I'm sure that potential further study could have been extensive it would be impractical to investigate a solution that has already been thoroughly researched, similar to light-mapping. Between the choice of studying PRT and understanding it to its full extent, or moving on to devise a new solution, I felt that the latter would be more beneficial.

6.3 (Ir)Radiance Cache-Based Methods

There are many global illumination techniques in use today. None of these techniques are capable of real-time speeds, yet some have been modified in such a way as to approximate in some way on a GPU. Photon mapping has been implemented on a GPU with some success by Larsen and Christensen (larse04). However, the most progress has been made in the area of real-time irradiance and radiance cache techniques. I found and selected three that were out of the ordinary in their approach to simulating (ir)radiance caching and I felt that they were potentially worthy of extra research. Unlike light-mapping and PRT, they were all also very unrefined methods so I saw plenty of scope for improvement and further investigation.

6.3.1 Radiance Mapping

The first of the techniques that I discovered that was worthy of investigation was one that was devised by a man named Nathan Vegdahl. In his article (vegda00), Vegdahl theorises that it would be possible to simulate diffuse inter-reflections in real-time by using hemi-cube maps or hemispheric maps.

He states that a hemi-cube or hemisphere should be rendered from the point of view of each vertex; the direction in which the scene should be rendered from is

[2] Affine transformations include translation, scale and rotation.

dependent on the vertex normal. The irradiance for any given point in 3-dimensional space is defined by the integral of all light that reaches that point, so his method works by rendering the scene for each vertex and classifying this as the sum of all indirect illumination for the scene. With this irradiance value interpolated across each face and combined with direct illumination, this has the potential to achieve not only pleasing but *correct* results for models of a reasonably high tessellation.

Also, Vegdahl's theory is a remarkably simple one. Factors such as light attenuation are already combined within the method: objects that are further away will be smaller, therefore contributing less light. However, despite the advantages of this technique I was worried about the performance degredation caused by rendering the scene once per vertex, but in comparison to the other techniques based on irradiance cache I am convinced that this model may be the fastest.

I decided not to research this method further as there was very little left to actually research. The technique is very simple to understand and the article written by Vegahl covers possible implementation problems and solutions, so there was little need to supplement it with any of my own research.

6.3.2 Instant Radiosity

Instant Radiosity is a technique that is quite old, yet also quite obscure. According to Nick Chapman (chapm04) the technique has existed since 1997, yet there is little evidence on the Internet to support his claim.

The reason for its obscurity becomes clear when reading about how the technique operates. It involves tracing rays from a light source in random directions. Where these rays intersect any geometry, a point light is spawned with a diminished version of its parent's intensity and colour. Then a ray is fired again from the previous intersection point and so on until a threshold is reached and the process is terminated. This is done several hundred times. This sounds like a reasonable pre-processing step, yet the most interesting part about the technique was that it spawns hundreds and hundreds of point lights. See Figure 3 for an example of this.

This model was clearly an unfeasible technique to cover, yet it was so inventive I thought it deserved some merit. I attempted further research on this topic with few results.

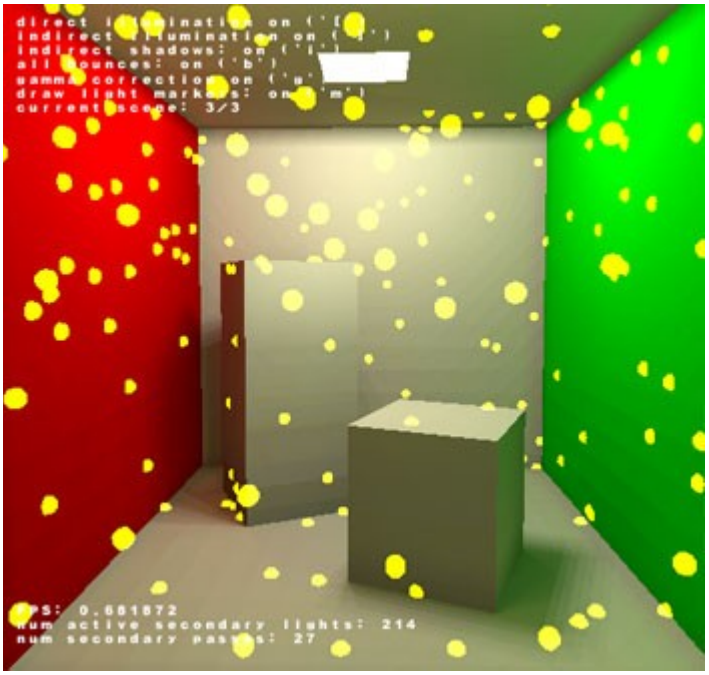


Figure 3 - Instant Radiosity

Image by Nick Chapman [chapm04]

6.3.3 Irradiance Cache Splatting

Irradiance Cache Splatting (ICS) is the most intriguing of all the (ir)radiance cache based implementations that I discovered. It utilises the same theory of irradiance cache, but in *reverse*. Instead of finding which lights contribute illumination to a specific point, this method works by finding what points are affected by a light. The specific details of the algorithm are covered in section 3 of Appendix B. Because of the unique approach to global illumination, this method fascinated me; I decided that this would be an interesting model to research further and attempt to implement.

Research for this method was from only one source which was a white paper written by Gautron, *et al* (gautr05) published for the Eurographics Symposium in 2005. The paper explained the principle of their reversed irradiance cache and also further implementation details. While this may have been advantageous in that there was only one definitive source for ICS information, it also meant that

any information that was missed from the paper was completely unavailable. Most of the time that was spent during the research phase of this method was deciphering the white paper and making sure that there were no holes. During project meetings, gaps in my information became apparent, and the week following the meeting would be dedicated to finding more information and filling the gaps.

After many weeks, the implementation details were eventually unravelled, but it became apparent long before that the method would not run at interactive speeds. Gautron *et al* claimed to have achieved real-time performance for global illumination using their algorithm on a nVidia Quadro FX 3400 PCI-Express graphics card on a machine with 3.6 GHz Pentium 4 CPU and 1 GB of RAM. However, by their own admission, the method only achieved an appalling 3 frames per second on a complex[3] model. It became obvious that this method was not suitable for further study and implementation.

However, once the method was fully documented and understood it was possible to take some of the concepts of ICS and apply them to a different global illumination model. The next section covers the final model that was conceived from Irradiance Cache Splatting: *Photon Splatting*.

[3] The white paper defines this as between 20,000 and 50,000 polygons.

7. Photon Splatting

7.1 Photon Mapping Overview

7.1.1 Introduction

Photon Mapping is a global illumination model created by a man named Herik Wann Jensen. His book, *Realistic Image Synthesis Using Photon Mapping* (jense01), describes a process involving the casting of *photons* – particles of light energy – into a scene from each light source. These photons bounce around the scene and eventually become stored in what he calls a *photon map*. A photon map is essentially a large special storage structure such as a *k*d-tree[4].

7.1.2 Generating the Light Data

For each light source, photons are cast out in random directions. Each photon is assigned a specific amount of energy which is based on the number of photons and the total energy (measured in watts) of the light source. When a photon intersects any geometry, it has a chance of being reflected, transmitted[5] or absorbed. When a photon is absorbed, it is stored in the photon map.

When a photon is reflected, however, the light bounces off the surface. The manner of this bounce is determined by whether it is a *specular* reflection or a *diffuse* reflection. A specular reflection is when the light is simply reflected as-is, whereas a diffuse reflection means that the photon absorbs the material's colour before being reflected. This gives us our *colour bleeding* effect.

7.1.3 Finding the Illumination

After all the photons are cast from each light source and stored, we need a way of finding the sum of all the surrounding photons' energy for a given point. This is where a *k*d-tree is useful: it allows us to search for photons in an area around a point in space. The maximum number of photons that are found depends on the radius of the search and also a possible cut-off value for large numbers of photons.

[4] A *k*d-tree is a form of spatial subdivision. Its basic function is to allow the storage of a very large number of points in *k*-dimensional space (3-dimensional, in this case) and a facility to retrieve these points quickly.

[5] Transmission means that the photon passes through a material. This is used for refraction or subsurface scattering, for example.

To calculate the illumination for each point from the photons found in its proximity we use this equation:

$$\underline{E} = \frac{1}{m} \sum \underline{\rho}(R - |\underline{p_k} - \underline{p_s}|^2) \underline{d_k} \cdot \underline{n_s}$$

Where **E** is the outgoing energy for a given photon, *m* is the number of photons collected, **ρ** (rho) is the energy of the photon, *R* is the search radius, $\mathbf{d}_k$ is the direction of the photon, $\mathbf{n}_s$ is the surface normal at the search point and $\mathbf{p}_k$ and $\mathbf{p}_s$ are the positions of the photon and search point, respectively. To obtain the final result we sum the energy of each photon, attenuating by the distance the photon is from the search point. This will give us our final illumination value.

7.2 Photon Splatting

7.2.1 What is Photon Splatting?

Photon Splatting is a new real-time global illumination model that I conceived by taking the principles of *Irradiance Cache Splatting* and combining it with Jensen's *Photon Mapping* model (jense01). Photon mapping can be much less computationally expensive than irradiance cache, and because of the poor performance of Irradiance Cache Splatting I have decided to use it instead. The specifics of the algorithm are covered in section 8.1.

Just like Irradiance Cache, Photon Mapping requires an expensive traversal of a large and complex data structure (generally a *k*d-tree) which is not possible on a GPU. The splatting concept is fundamental in replacing this expensive process with a GPU-compatible solution.

7.2.2 Photon Splatting versus Irradiance Cache Splatting

The same framework still exists in Photon Splatting as for Irradiance Cache Splatting. I refer to each Photon as a 'record' – as in an 'irradiance record' – which is a point in 3-dimensional space with information about the irradiance at that point. Each photon is given the opportunity to calculate lighting data for each of the pixels for which it contributes illumination, just like for ICS. The photons could be distributed in any fashion, but I am currently using a "photon

per vertex" method for demonstrational purposes. Possible alternatives to this distribution are given in section 14.2.

The main reason for replacing the model is related to the speed of the implementation, as previously stated. As shown in Listing 3, the record computation is a simple one. However, irradiance cache requires a more complex model to yield correct results.

7.2.3 The Faster Model

Listing 1 - The Algorithm Skeleton

```
For Each Record k = { Pk, Nk, Ek, Rk }
Do
        Calculate Quad For Point Pk with Radius Rk
        For Each Pixel f = { Pf, Nf, Tf } in Quad
        Do
                // Calculate the distance from
                // the fragment position and the
                // record position
                D = Pf - Pk
                R = |D|

                // Calculate the weight and irradiance
                // Add to the splat buffer
        End
End
```

As you can see from Listing 1, the basic outline for each algorithm is the same. However, the weight and exit radiance contribution calculations for each model vary in size significantly. They are given side by side for comparison purposes on the next page in Listing 2 and Listing 3.

At first glance the listings provide little evidence as to which is the more complex operation. However, the ICS algorithm includes the rotational and translational gradients, ∇_r and ∇_t, which are based on the original irradiance cache algorithm. These values ensure that the correct irradiance values are obtained from light that is striking a surface at extreme incident angles. To calculate ∇_r and ∇_t, we use the rules of curl (Figure 4) and divergence (Figure 5) specified by the nabla[6] (∇) symbol (mathwb).

[6] Also known as "del".

$$\nabla \times \mathbf{F} = \begin{bmatrix} 0 & -\frac{\partial}{\partial z} & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial z} & 0 & -\frac{\partial}{\partial x} \\ -\frac{\partial}{\partial y} & \frac{\partial}{\partial x} & 0 \end{bmatrix} \mathbf{F}$$

Figure 4 - Curl [mathwb]

$$\nabla \cdot \mathbf{F} = \frac{\partial F_x}{\partial x} + \frac{\partial F_y}{\partial y} + \frac{\partial F_z}{\partial z}.$$

Figure 5 - Divergence [mathwb]

Listing 2 - ICS

```
// Calculate the weight
w = 1/((R / Rk) + sqrt(1-nf·nk))

// Ignore any pixels that don't
// have enough weight
if ( w < 1/a ) Continue

// Compute the contribution of
// the record k at point p
E = Ek(1 + (nk×nf)·∇r + D·∇t)

// Compute the outgoing
// radiance and accumulate it
S(x,y).rgb += wE × Ef
S(x,y).a += w
```

Where *a* is a user-defined accuracy value (see section 13.3.5) and where ∇_r and ∇_t are defined as the rotational and translational gradients, respectively.

Listing 3 - Photon Splatting

```
// Calculate the distance from
// the centre
d = |( Tf * 2 - 1 )|

// Ignore any pixels outside of
// this radius
if ( d > Rk ) Continue

// Calculate the angle of
// incoming radiance
i = Nf · -D

// Calculate the angle of
// outgoing radiance
o = Nk · D

// Calculate the distance
// attenuation
A = 1 - ( R / Rk )

// Calculate the final
// attenuation using inverse
// square and the circular
// fall-off, d
w = i * o * A² * ( 1 - d )

// Accumulate the attenuated
// incoming radiance
S(x,y).rgb += Ek * w

// Accumulate the weight
S(x,y).a ++
```

To determine which algorithm is faster, the sum of the number of shader operations that the GPU must perform for each fragment is found. The result of this tally yielded that Photon Splatting was indeed the faster method, as shown in.

Table 1 - Shader Operation Comparison

	ICS	**Photon Splatting**
Arithmetic Operations	53	34
Square roots	2	1
Branch Statements	1	1

The Photon Splatting model, while less accurate, is faster and therefore more suited for real-time applications.

8. From Paper to Pixels

This section covers the entire development process: it describes the design and implementation phases that took place. A description on my approach to design is provided, where I will justify the decisions made when planning the implementation. This is followed by an overview of the implementation phase, where the analysis and design are put to use.

8.1 Approaching Design

The purpose of designing an application is two-fold. The first purpose is to set goals with clear and concise targets for the project. Without this, the project could potentially spiral out of control as the overall scope of the project would become bloated and meaningless. The second purpose is to identify possible problems before they occur so that solutions can be found without having to re-write large sections of code.

8.1.1 Middleware?

There was some deliberation over what engine the application should use. One advantage of using middleware was that it would shorten development time, but it would also mean learning the particular methodology and intricacies of the API. This is one argument in favour of writing an engine from scratch: worrying about how the engine works is not an issue. Writing the software would also be beneficial because the underlying application would be completely designed and built to support the photon splatting algorithm. This would mean less bloat from unnecessary features and more speed. I determined eventually that I would write the software using libraries that I had previously written.

8.1.2 Setting Goals

The design of the photon splatting demonstration evolved from the guidelines set down by the project specification (Appendix A, sections 12.1 and 12.4). The goal of the application was to clearly demonstrate that the algorithm worked correctly and show that global illumination was possible on a GPU, and these became the guiding principles when writing the implementation. Another goal was that it must run at interactive frame-rates. This is actually two goals: it must run at a

real-time speed and also feature an interactive element to prove that it can be used for an interactive simulation.

After the list of goals was established, it was necessary to determine how these goals would be achieved. A few guidelines had to be set, such as: what platform to write for; what hardware it should run on; what programming language was it to be written in; and how it would be able to demonstrate the graphical algorithm at real-time speeds. The following was concluded:

- To be be written in C++.
- Must run on Microsoft® Windows®.
- Will use Microsoft's DirectX SDK as the underlying graphical API.
- Should run on at least a 2GHz CPU with 1GB of RAM and an nVidia GeForce 6600.

The hardware that the application supports was determined by both the hardware available and also by the algorithm's complexity. The algorithm is complex and was therefore predicted to run only on recent generation hardware, so a moderately new system was selected as a basis for control. However, one of the purposes of the project was to demonstrate that global illumination was possible on affordable hardware, so the system selected was deliberately not a high-end machine.

After the goals had been set, it was time to plan the implementation itself. The application was designed in such a way as to feature a shader-based 3D renderer and a facility for basic user input and no more. The user interface design and implementation was quite simple: the user merely has to press the left or right arrow keys to rotate the orbiting spheres around the central spheres. The API for user input was done using the Windows API. However, the rendering process of the application was significantly more complicated. It is covered in the next section.

8.1.3 Designing the Renderer

The renderer is the largest and most important part of the application. Obtaining a correct and efficient design was crucial to prevent problems arising that would

have been impossible to avoid. The entire rendering process had to be efficient and streamlined, but also easy to maintain. Because the graphical API choice was one that was already object oriented, some of the process was simply being able to correctly encapsulate Direct3D's COM interfaces. However, the actual *procedural* aspect – *id est.* the order of execution – had to be designed to ensure that everything was done in the correct order.

The features that are needed for photon splatting could be deducted from the algorithm itself. The analysis and design of the algorithm in Appendix C, section 14.3, immediately shows that we need a *splat buffer*, a handful of pixel shaders and a fast view-aligned quad[7] rasteriser.

The *splat buffer* was actualised as a render target texture of a reasonably small size (as detailed in section 14.4.2). When calculating the irradiance and weights in the pixel shader, the per-fragment position and normal were also needed. Therefore it was necessary to also use render targets for the *position buffer* and the *normal buffer*. These buffers were combined together to produce the final scene in a pixel-shader by rendering a full-screen quad.

The renderer was built from the start to support shaders in the form of Direct3D Effect files. There are only two types of vertex shader in the application: one for rendering normal geometry that could be rotated, translated and scaled, and the other for rendering screen-aligned quads of any given size. There were also only two types of pixel shader within the application: a material shader and a post-process shader. A material shader was bound to specific geometry and determined how that geometry was shaded. A post-process shader was a material shader assigned to a full-screen rectangular polygon (hereby referred to as a "quad"), allowing post-process effects such as combining two previously rendered passes. The only shader that was an exception to both of these cases in some way was the splat shader.

The splat shader was a special case as it was required to be bound to a list of billboards. To achieve the splatting effect the scene was rendered normally followed by a *splat pass*, which involved rendering a billboard for each vertex of

[7] Also known as a "billboard". This is a square polygon that has been rotated to face the camera, no matter where the camera is in relation to it.

the geometry. This was when the splat shader was active. The billboard renderer was an extension of the standard geometry renderer: it still accepted a vertex buffer as an input argument and stored a pointer to this data. When it came to rendering, however, it did not pass this vertex buffer to Direct3D for rendering but instead iterated through the list of vertices stored in the vertex buffer and rendered a quad of user defined dimensions to the screen for each one. This quad was oriented such that it always faced the camera (using the inverse of the view matrix) and was also centred on the position of the vertex. This was done by passing a customised world matrix based on the vertex's position and the object's rotation to the vertex shader. It would have been advantageous to have this process done by the GPU, but unfortunately one constraint of vertex shaders is that no new geometry can be added so the addition of the full screen quad for each vertex was impossible.

Finally, a system was implemented to ensure that the scenes were rendered in the correct order to the render targets. This system involved a list of render targets coupled with their appropriate scene objects and for each frame this list would be rendered. Scenes that were dependent on a previous target were given a reference to this target and added to the end of the list to ensure that it was rendered after its dependency.

8.2 Implementing Photon Splatting

8.2.1 How Analysis and Design Helps

When implementing a complex or large system, an analysis and design phase is crucial to ensure the system's stability and maintainability. A good analysis and design will cover every aspect of the system including many of the system's potential problems while also maintaining ease of development and be schedule efficient.

As the system's goals were already outlined, it was a simple process to translate these into a package of components and procedures. Each component was then analysed to give a set of sub-components and so forth until each system had been broken down into its discrete parts. All that remained was to implement each of these areas.

To do so, I translated each "component" into C++ classes. The member functions and data of these classes depended on how the class was used, which was dictated by the design. For large or complex classes or subsystems, I drew out class diagrams in UML[8] to help me visualise how they worked with each other. The flow of execution was written out using data flow diagrams so that there was no confusion when it came to deciding what order a sequence of steps should be executed in.

This process also allowed possible problems to be identified early. For example, when designing the render targets I had inherited them from a texture interface; it became apparent that this was convenient yet very unwise. It became clear that overloading the texture interface would mean that I would run into a problem called *diamond inheritance*[9]. The only way to have fixed the problem, had I not planned ahead, would have been to either re-write the renderer or to duplicate the entire texture class code.

Another potential problem was due to the sampling of buffers with the GPU's texture filtering capability. While this produced a nice dithered effect on standard textures, it invalidated normals and positions when sampling the normal and position buffers. The way to solve this was to specify that these buffers should be sampled with linear texture filtering disabled.

8.2.2 Building the Implementation

The implementation was a simple process once the algorithm had been designed to a satisfactory level. It involved simply translating the designed classes and procedures into C++ code. The layout of the objects and the way that they interacted had already been done in the design, so getting the skeleton of the code written was fairly simple. However, the task of filling the skeleton code was less simple if the design was not sufficient in that area. When I found this to be the case, I went back to the design and restructured it and made sure I had an adequate solution for every possible problem that I thought might arise. The design and implementation was a very *iterative* process, as the unexpected problems faced in the implementation would affect the design.

[8] Universal Mark-up Language: allows the communication of technical ideas in diagrammatical form.
[9] This occurs when two subclasses inherit from a parent class, and then another class inherits from both of these classes. If represented diagrammatically it gives a diamond shape.

As the design was developed using UML, the natural object orientation that UML provides helped considerably when writing the C++ code. Each object was suitably encapsulated and interfaced. Each component – a package of classes grouped by context, such as a renderer – was given its own C++ namespace, which improved code readability.

The two main components identified by the design were the renderer and the user interface. The user interface was implemented using the Windows API and its in-built keyboard functions. It simply allowed the user to hold the left or right arrow keys to rotate the orbiting spheres clockwise or counter-clockwise. Also, the implementation had to provide a way for the user to switch render targets for debugging purposes. The renderer's main function was to provide a simplified wrapper for Microsoft's DirectX API which allowed easy access to textures, meshes and shaders. These complex components, once established, allowed the easy implementation of the splatting algorithm.

The implementation was built, as specified, using Microsoft Visual Studio .NET 2003 on Microsoft Windows XP. The machine used was an Athlon 3000+ 2GHz CPU with 1GB of ram and an nVidia GeForce 6600.

8.2.3 Writing the Algorithm

This algorithm itself was written completely for the GPU using Microsoft's High Level Shader Language (HLSL). Using the pseudo-code given in Listing 3, it was a simple matter to translate this to HLSL. Therefore the GPU was dedicated to rendering the scene and processing the algorithm. However, the first version of the solutions involved a per-frame pre-process on the CPU which prepared the scene and data for the algorithm to run.

As I have previously mentioned, each vertex had to be replaced by a view aligned quad. Each frame the Direct3D vertex buffer had to be unlocked, processed and re-locked. While the buffer was unlocked, the renderer had to prepare and render a quad using the splatting shader. The first version had a design flaw that forced the implementation to use CPU time for moving the quad's vertices out of view space. However, this area was redesigned and the matrix transformations were done entirely on the GPU and have therefore improved performance dramatically.

8.2.4 Correcting Design Flaws

It is impossible to accurately predict what problems and set-backs one will face when writing a design, so I was aware that designing *too much* could and would be detrimental to my schedule. A lot of the designed components would become obsolete once various issues were encountered. I therefore used a very flexible and layered method of design: I designed the top level functionality, descending down until an appropriate level of design for the task had been reached. I determined this by the complexity of the task or how much knowledge I had of implementing it. For example, areas which were very complex were designed fully, whereas simple tasks were simply summarised. Also, areas in which I had little knowledge were not designed fully until I had adequate knowledge in which to do so. Some of these areas are listed in the following sections.

8.2.5 Learning the DirectX API

Most, if not all, of the problems that were unexpected were related to either the use of Microsoft's DirectX API or the graphics hardware itself. Specifically, the problems occurred from my misuse of the API. I had never written a shader-based renderer before, nor had I written a renderer in DirectX, so this was quite a steep learning curve for me. Previously I had only written my own renderers or I was using an API that was significantly more compact (and not real-time), so I had intricate knowledge what functions of the API precisely did. However, in Microsoft's DirectX there was a significant amount of work done for me, but problems arose when it did things that I was not aware that it had done.

Frequently this was the case when I lacked sufficient knowledge of the feature that I was using, or I was forced to use a different feature to do the work of one that was missing. To clarify the latter point: occasionally I would realise that a certain feature that I required of the hardware did not exist, so another feature had to be used to approximate the function of the missing one. For example, I was unable to pass the position of each fragment to the pixel-shader, so I was forced to use a texture coordinate register instead. What I forgot to account for was that these values are automatically interpolated and this caused problems when passing the photon position to the pixel-shader. Eventually I restructured the design and used a pixel-shader constant.

8.2.6 An Example Problem

The most vexing problem that I encountered in this project was the issue of the off-screen buffer values being automatically clamped. These buffers are intended for display so any values written to them are clamped to a displayable colour value in the range [0,1]. This caused terrible problems in two situations. The first was when the splat weight is accumulated:

Listing 4 - Accumulating the Weight

```
Out.rgb = saturate( angle*atten*atten ) * vColour * D;
Out.a = 0.001;
```

As shown here I had to reduce the weight increment to 0.001 instead of a simple unit integer increment. When it came to the normalisation of the splat buffer, I multiplied by 1000 to obtain the correct result:

Listing 5 - Normalising the Buffer

```
float Count = Splat.a * 1000;
if ( Count > 1 )
{
	Count = saturate( 1 / Splat.a );
	Splat.rgb *= Count;
	Splat.a = 1;
}
```

However, the more serious issue involved the use of the position buffer. In this buffer the world space positions of the fragment were stored so that the splat shader could obtain per-pixel information on the depth and distance from the photon position. However, world space coordinates were automatically clipped to a range of [0,1], which proved very problematic. After failing to find any information on 3-channel de-normalised floating point buffers, I instead had to resort to storing the positions in *projection space*, which provided me with a normalised [x,y] coordinate in the range [-1,1] which was then scaled to [0,1] and a depth value in the range [0,1].

This, unfortunately, resulted in a large loss of accuracy and caused horrible problems such as the splat shader picking up on background pixels and shaded them incorrectly. Instead I adopted a quick solution, which was that a type of "world-space normalisation" would take place as a pre-process, where the world was scaled uniformly to a [-1,1] range on each axis. Little accuracy was lost in this process.

8.2.7 Mesh Tessellation

This method applies to photon splatting using a photon-per-vertex approach. In an arbitrary scene, vertices will be distributed in a way that would produce inefficient and often incorrect results. For example, a blue sphere is placed near a white wall. The blue sphere's photons will project a bluish tint onto the white wall, but the wall will be unable to project any light back. This is because the vertices of the wall are too far away from the sphere. To solve this problem, the wall needs to be uniformly tessellated so that photons are present to cast light to their surroundings. There are numerous algorithms for tessellation available, but in some cases it may be undesirable to tessellate every surface. For example, perhaps the modeller of the scene does not *want* a particular surface to emit light. The scenes used in the demonstration application were manually tessellated using 3D Studio Max.

Also consider, however, that an object can have too many vertices. The photon splatting works by calculating outgoing radiance for each fragment within a radius. This means that any photons that are too close together are contributing to the same fragment and because their contributions are then normalised, it will then only be seen as one contribution. We are performing two splat calculations for just one fragment so; a mesh must only have one photon for every half-radius distance to maintain uniformity.

9. Evaluation

After any project is completed, it is necessary to step back from it and analyse it from both a critical and appraising points of view. This process helps prevent future projects from suffering from similar problems and reveals strengths that should be included in all projects. This section provides a breakdown of the project's strengths, weaknesses and areas for improvement, including extensions to the algorithm and optimisation techniques. Finally, a summary of the knowledge that I have gained from this writing this project is given.

9.1 Successes

9.1.1 Research Strengths

It is difficult to accurately gauge the "successes" of research as there are few obvious ways that research will directly affect the design or implementation phases. However, it can be said that the research performed concerning the four techniques was useful because it helped give an accurate overview about how current solutions worked. Without this knowledge it would have been far more difficult to devise a global illumination technique.

After the initial research was done, a method was selected to investigate further. This was *Irradiance Cache Splatting*, and in order to be of any use, it had to be broken down and completely understood before any design could be started. When this had been done it was simple to take the method's key aspects and apply them to a simpler model. ICS was selected from the four principle methods researched, not only because of its unusual approach to accomplishing global illumination, but also because of the results that it achieved compared to the other solutions. Despite the excellent results, I saw room for much improvement for this method so I preferred to study this technique further instead of one of the other, more tried-and-tested techniques.

9.1.2 Design Strengths

The extensive research that was done for ICS was used to create a design based on ICS's core strengths, while hopefully removing as many of the weaknesses of

the technique as possible. The strengths of ICS were applied to a more simple global illumination model: photon mapping.

The simplicity of the model worked greatly in my favour. The original ICS algorithm was not overly complicated, but because of the way that the original white-paper was written it was difficult to decipher exactly how it worked. Despite the simplicity of the model, it was still necessary to cut the model down as much as possible. The shader operations used for the splatting process were of an acceptable quantity for PCI-Express hardware (see Table 1 - Shader Operation Comparison), but it became advantageous to cull as many of these as possible for non-PCI-Express hardware.

The most pleasing result of the algorithm's design was that it requires absolutely no pre-processing to function. Unlike nearly all of the other methods investigated, this method has no lengthy step before execution that will generate data for each mesh. This method was an excellent choice because allows geometry to be completed deformable, giving opportunities for animation and other dynamic object interactions. It was also useful because there is no pre-processed data to store or load at any time. This is where photon splatting is truly superior to all the other methods such as light-mapping and pre-computed radiance transfer.

I allocated a significant amount of time to perfecting the design. Many people that I sought advice from regarding my schedule disagreed with this decision, including my project Supervisor, but I felt that it was necessary to have the algorithm completely documented and designed before any implementation was created, even if most of the information would not make it to the report. I stand by this decision, as I feel that I was rewarded for it. However, during the design of the algorithm I was also implementing the framework upon which the application is built. Once the design was complete, the photon splatting implementation only took 11 days.

9.1.3 Implementation Strengths

The implementation was also a great hurdle to overcome. The development of the application started in December, but it was only in late February when the actual algorithm began to take shape. Between early December and early

February, I had to write a 3D shader-based renderer that was capable of displaying the algorithm. Having never written a shader renderer in DirectX before disadvantaged me highly, so I had to learn the intricacies of Direct3D and also learn Microsoft's HLSL. I mention this in the context of the project's strengths because I feel that I have successfully achieved this goal while keeping on schedule and also successfully implementing the algorithm. In order to use my time effectively, I also designed the algorithm parallel to the framework's implementation.

I feel that the application itself is also a success. It clearly demonstrates that the algorithm does indeed produce diffuse inter-reflections (as shown in Appendix D). Not only this, but it runs at interactive speeds on affordable current generation hardware, which is the most important goal of this entire project. The application is also interactive, allowing the user to rotate the spheres around the centre sphere, demonstrating that the algorithm is not a pre-computed method such as PRT or light-mapping.

Finally, the algorithm is capable of producing results comparable to a low resolution offline global illumination implementation. This is excellent because I had expected it to be capable of simple diffuse inter-reflections, I did not expect it to produce results of such good quality. It is possible that the quality could be further improved to match a full resolution offline photon map, but I do not believe that today's hardware would be capable of handling this just yet.

9.2 Failures

9.2.1 Research Weaknesses

The one fundamental problem with the research phase was that I was not able to cover the enormous wealth of information related to the topics in such a short period of time. The information for light-mapping, while extensive, was not difficult to find as it has been an established technique for many years now. However, PRT, (Ir)Radiance Caching and particularly the area of Spherical Harmonics were vast topics that were more obscure. This presented me with a difficult choice: I had to either cut down on the amount of information that I had to cover, or extend the schedule to allow me to finish documenting them. I chose

to compromise and extend the deadline for each by a week while reducing the amount of material that I intended to cover.

This means that while plenty of information is presented in Appendix B, only a small portion of the research is actually documented there. It also means that the design and implementation phases had to be moved, allowing me less time to complete the algorithm. The research phase was intended to last three weeks, with a further two to complete the documentation. In the end it took a total of seven weeks.

9.2.2 Design Weaknesses

Despite all of the successes of the design process, the design itself was deliberately flawed in various ways. Knowing that the algorithm was based on one that was already only running at 10 frames per second on PCI-Express hardware, extreme optimisation methods were needed to make it perform at 25 frames per second on *non*-PCI-Express hardware. Commonly optimisation techniques lead to inaccuracy and unfortunately this was the case. However, a lot of the optimisation methods, while producing *scientifically* incorrect results, still proved to be visually acceptable.

There is, unfortunately, one side-effect of any splatting-based global illumination technique. When splatting a record to the splat buffer, all pixels within the splat radius are processed and these pixels pass through two checks before being shaded. If either of these tests fails, the pixel is not shaded and the photon's contribution for this pixel is zero. The first of these tests obtained the distance from the pixel's position to the record. If this distance is beyond the splat radius the pixel was rejected. This test is followed by the normal test. If the pixel's normal was facing in the same direction as the photon's direction then the pixel was rejected as no illumination reaches the surface. However, the problem arises when a given pixel was within the range of the splat, was then visible to the camera, was facing in the correct direction and yet there is geometry between the photon and the pixel. Under realistic circumstances a pixel under these conditions would not be affected by the photon, but unfortunately there is no feasible way to check that the pixel was occluded.

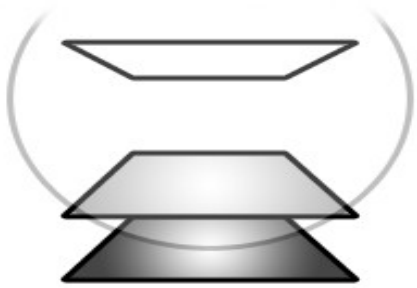


Figure 6 - Incorrect Splatting

The bottom plane receives illumination from the photon, even though it is occluded by the plane above it.

9.2.3 Implementation Weaknesses

Despite the successes of the design, some areas of the algorithm were deliberately approximated for speed. One such approximation is when calculating the outgoing radiance for a photon. The design states that the outgoing radiance for a point should be given by the integral of all incoming light sampled from a hemi-cube map (perhaps using a Monte-Carlo Integration[10] method). The hemi-cube map should be rendered from the photon's position using the photon's direction. However, the implementation phase was limited by time constraints and therefore cube mapping was not implemented. Instead a loose approximation is its place. The implementation still demonstrates diffuse inter-reflections and is still interactive; however, the diffuse inter-reflections are not always completely accurate. A side effect of this fix is that occasionally the sampler can "miss" a photon, giving the irradiance for geometry in the background resulting in incorrect colouring of a splat. However, once the splat buffer is normalised this effect is negligible.

The application meets the criterion of running at a minimum of 25 frames per second. However, this is for a scene consisting of, on average, fifteen to eighteen thousand polygons. It is probable that current generation applications will require much more than this. Also, this figure includes faces that are back-face culled, meaning that they are not visible as they are facing away from the camera. Photon splatting also has the disadvantage that vertices cannot be back-

[10] Monte-Carlo Integration involves taking random samples for a given domain to approximate mathematical integration. In this context it means that the hemi-cube will be sampled using a random direction based on the photon's normal.

face culled in the splat pass, otherwise colours will bleed into areas that they should not. This means that the polygon count, if increased, will reduce the frame rate quite substantially. However, this does not doom the algorithm: many possible optimisations can be employed to speed up the process. See section 9.4 for more details.

The most frustrating of all the problems encountered was related to my model format. I was converting to my own format from 3D Studio Max .3ds files, which unfortunately does not provide normal data within the file itself. Instead I had to use a common algorithm to generate them, which was fine for all kinds of polygonal geometry but failed to generate correct normals for spherical objects. After much investigation I decided to swap my implementation for a 3rd party library called *lib3ds*. This provided an automatic normal generation, but unfortunately it seemed to take into account the Max-assigned "smoothing groups", giving me the results you can see on page 75.

9.3 Improvements

The algorithm that has been provided in this document is still only the basic skeleton of something that could be much more complex. Currently it exists in a form that has plenty of opportunities for improvement. Some of these improvements could be purely for optimisation and are detailed in the following section, section 9.4. However, other improvements can be made to improve the quality and/or accuracy of the output. Of course, because we would be increasing the quality or accuracy, we would lose more processing time. The ideas presented here are therefore purely theoretical.

9.3.1 Variably Size Splats

One such improvement is using variably sized splats. For example, the size of the splat quad could be related to the power of the photon. This would mean that photons with more energy will travel further than photons with very little power. This method has the unfortunate effect of increasing the fill-rate of the splat buffer, so this is a technique that should only be used in conjunction with heavy optimisation.

9.3.2 Floating-Point Buffers

It may be possible to use a particular buffer format for the position, normal and splat buffers that would prevent them from being clamped to a [0,1] range. These buffers, called *IEEE floating point* buffers, would alleviate the need for the scaling process that is done in the current implementation. As we are currently scaling the world to a [-1,1] range or using the projection-space coordinates, we are losing quite a lot of accuracy as floating point buffers on a GPU are only precise to three to six decimal places. Therefore, writing to a floating point buffer would mean that we save CPU time by removing the scaling process and increase accuracy.

9.3.3 Changing the Photon Sampling Method

As shown in section 8.2.7, alternative photon sampling methods can exist for generating photons other than using a photon-per-vertex approach. It may be possible to use two different mesh formats. For example, one format can hold the true geometry, while the other could hold a list of photons that are bound to this geometry. However, this adds the need for an extra pre-processing step which this method has avoided so far. For the sake of speed this is not an unworthy trade.

Alternatively, photons could be sampled using a texture map. A texture could be applied to geometry that defines the colour and precision of photons at that point. To clarify, a red texture map could be applied with an alpha channel that varies over the surface of the image. A pre-process at loading time could be performed where this texture is analysed to determine the accuracy of the photons at each point. For example, if the alpha channel is high, more photons are added in the area surrounding that texel[11] in world space. However, if the alpha channel is low, few or no photons are added.

9.4 Optimisations

Global illumination has always been a slow process. Even when researchers have attempted to reduce the process down to real-time speeds, it is still not fast enough to be deemed "interactive". Photon splatting, while considerably faster, is no exception.

[11] Texture pixel

9.4.1 Performance Hits

The splat rendering process is the most expensive process of the entire application. For each vertex – of which there are tens of thousands – a screen-aligned quad must be rendered and blended with the background.

When processing the scene the splat renderer must render *four* times the number of vertices in the scene: for each vertex, the renderer must create four vertices for each corner of the quad. Each of the four vertices must undergo two extra matrix transformations to orient it with the image plane. This would not be too much of a dramatic performance hit for a simple scene, but for a highly detailed scene this would have a dramatic impact.

Also, when the quad is being shaded in the pixel shader, the weight and colour is computed. This requires sampling from the position buffer, the normal buffer, and a cube map as a worst case scenario. This process is *extremely* slow. Also, all the values calculated here are added to the existing values already in the splat buffer. This unfortunately requires a read from the buffer, which is still a slow operation on GPUs due to the nature of its onboard memory.

Therefore, in order to gain extra performance from the application, approximations in the *Irradiance Cache Splatting* model had to be made, transforming it into *Photon Splatting*. However, even here the process of rendering tens of thousands of blended polygons is something that graphics processors find tough. In this section I propose a number of methods that should improve performance of the solution.

9.4.2 Selective Photon Splatting & Combining with Other Models

It could also be possible for only certain objects to cast a diffuse reflection. For example, a plane with a dark texture would probably not reflect or absorb any diffuse inter-reflections, so it can be omitted from the splat pass entirely. This wall can instead be lit by a pre-computed method such as light-mapping.

9.4.3 Using a Lower Resolution Splat Buffer

The most obvious optimisation, perhaps, is to decrease the resolution of the splat buffer. This would decrease the fill-rate of the splatting considerably, but unfortunately resolution is lost.

In the application developed alongside this project, the splat buffer was created with a resolution of only 320x240, which was a fraction of the back buffer size. Resolutions lower than 320x240 were tried but unfortunately the quality had degraded too much to be of any use. In the output images it may be possible to see that the irradiance values from the splat buffer appear more pixellated than the rest of the image. This would be due to the low resolution.

9.4.4 Frustum Culling

A more efficient method of reducing overhead is *frustum culling*. This means that any photons that are not within the viewing pyramid are not processed. Frustum culling allows us to only process photons that are within the view and therefore ruling out any vertices that will not contribute to the scene.

However, if photons just outside the border are clipped they will not contribute irradiance to any points near the edge of the image. This is undesirable. Instead it would be more appropriate to use a bounding sphere around the photons using the splat radius to determine if the photons contribute any irradiance to pixels inside the frustum. This has the side-effect of including photons that may not contribute anything anyway, but this is an acceptable risk.

This process should also be done with photons that are just behind the viewport. If a photon is just behind the camera it will contribute illumination to any objects that are near the camera's view. Therefore these photons need to be accounted for. However, it may not be necessary to apply this process to the photons just beyond the far clipping plane. This is because the diffuse inter-reflections from photons at this distance are unlikely to be noticed.

9.4.5 Space Partitioning

It would also be useful to selectively process photons in the world using a space partitioning scheme such as a *k*d-tree, oct-tree or binary space partition (BSP) tree. This would cull off any photons that were not visible from the scene, while also allowing us to order the splatting process on the z-axis (because depth buffering is disabled on the splat pass).

It would also provide the added benefit of only rendering the geometry in proximity to a particular group of vertices. For example, take a scene that consists of two rooms and one of these rooms contains a pedestal. The pedestal is close to a wall, so there will be diffuse inter-reflections occurring here. It would be useful, when processing the photons in this area, to completely omit the empty room from processing while casing a diffuse inter-reflection from the pedestal. The empty room will still course still be drawn normally, but it will not be drawn in the splat pass.

Alternatively, a *k*d-tree could be used to find geometry within a particular radius of the mesh that is casting the inter-reflection. This could be accomplished by creating a bounding sphere around all the photons for that mesh, adding the splat radius to the radius of this sphere and testing which geometry intersects with this sphere. However, this process could be potentially wasteful of CPU time for complex scenes and is probably not worth considering.

9.5 Knowledge Gained

Over the course of this project I have learned many new things. The peculiarities of the GPU became obvious after much aggressive debugging and I eventually completed the algorithm using Microsoft's HLSL; a language that I had never used before.

I have also gained knowledge in other global illumination models. The research phase revealed many possible avenues for investigation, but I selected the more appropriate methods that I discovered based on their obscurity, or my personal interest. The knowledge that I gained here allowed me to adapt a current, yet obscure, technique for global illumination into a viable solution for real-time graphics.

This research also introduced me to many mathematical concepts that I was either not or only preliminarily aware of. The PRT section opened a door to a topic called *spherical harmonics*, which is a technique that is based on many complex mathematical theorems. I had also never studied Irradiance Caching before, though I had a basic knowledge of the principles involved. I would now

be confident in implementing either of these after a period of further research and design.

Finally I feel that I have learned valuable skills from organising the project. I have learned that my skills in organisation were not quite what I thought they were, and I was forced to get organised within the first few weeks of this project's inception. I have since then been able to accommodate for all kinds of schedule alterations, including weeks where a personal deadline was not met or unavoidable circumstances prevented me from doing any work for a period.

I have also learned how to research and record material effectively. When investigating the methods in the research phase I was dealing with large amounts of information that was sometimes only partially useful. This information had to be efficiently filtered and organised, which I feel I have done successfully.

10.Conclusions

Overall the project was a success. I have researched four other methods of global illumination and documented them appropriately, outlining the process of implementing each one. I have also successfully implemented a global illumination phenomenon on a graphical processing unit that runs in real-time. It also conformed to all the guidelines that I established in the specification. The implementation works correctly and at interactive speeds. Furthermore, I have completed these tasks within the given time frame, despite schedule slippages.

In *Research Methods* I have clearly identified my methods for collating and organising any data that I obtained. Most of this information was from on-line resources as there was little physical media that existed to help me in my investigation. *Current Solutions* looked at the product of this research in which I documented my findings and analysed each method's suitability for a GPU global illumination implementation.

This was followed by the final algorithm's development. *Photon Splatting* detailed the inception and design of the final algorithm that was based on a technique called *Irradiance Cache Splatting*. This process was then taken a stage further in *From Paper to Pixels*, which was a summary of the implementation process, including how all the research and design came together to form the final solution.

Finally the entire project was evaluated in the *Evaluation* section. The strengths and weaknesses of each stage of the project were identified and possible improvements were given. A large portion of this section was dedicated to optimising the technique, as despite the algorithm's performance in comparison to other existing techniques, it is still slow.

I am satisfied this project meets all the standards that I set out to accomplish. There is a lot of room for further optimisation and extension, but the primary goal was to research techniques and implement one rather than to produce a single technique that excels. In the future I hope to extend this research to cope with

the extensions and optimisations that I have listed in sections 9.3 and 9.4 and to also extend it to other global illumination phenomena.

I hope that you have found this an interesting proposal for global illumination. While the final technique itself may sound unconventional in its approach to global illumination, I believe that I have demonstrated that it is not only an effective technique, but also a fast one when compared to existing solutions.

Thank you for reading.

End.

11.References

[mille99] Kurt Miller *Lightmaps*
http://www.flipcode.com/articles/article_lightmaps.shtml

[hodor01] Luke Hodorowicz *Advanced Lightmapping*
http://www.flipcode.com/articles/article_advlightmapping-pf.shtml

[chann03] Keshay Channa *Light Mapping – Theory and Implementation*
http://www.flipcode.com/articles/article_lightmapping.shtml

[nuyden] Tom Nuydens *Radiosity Lighting*
http://www.delphi3d.net/articles/viewarticle.php?article=radiosity.htm

[sloan05a] Peter Pike-Sloan, Ben Luna, John Snyder
Local, Deformable Pre-computed Radiance Transfer
http://research.microsoft.com/~ppsloan/ldprt_final05.pdf

[sloan05b] Peter Pike-Sloan, Ben Luna, John Snyder
Pre-computed Radiance Transfer for Real-Time Rendering in Dynamic, Low Frequency Lighting Environments
http://research.microsoft.com/~ppsloan/shillum_final23.pdf

[green03] Robin Green *Spherical Harmonic Lighting: The Gritty Details*
http://www.research.scea.com/gdc2003/spherical-harmonic-lighting.pdf

[msdna] Microsoft *Pre-computed Radiance Transfer (Theory)*
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/advancedtopics/PRT/PRT.asp

[msdnb] Microsoft *Pre-computed Radiance Transfer (Equations)*
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/advancedtopics/PRT/PRTEquations.asp

[wikia] Wikipedia *Associated Legendre Polynomials*
http://en.wikipedia.org/wiki/Associated_Legendre_polynomials

[wikib] Wikipedia *Spherical Harmonics*
http://en.wikipedia.org/wiki/Spherical_harmonic

[mathwa] MathWorld *Legendre Polynomials*
http://mathworld.wolfram.com/LegendrePolynomial.html

[mathwb] MathWorld *Nabla/Del – Curl and Divergence*
http://mathworld.wolfram.com/Nabla.html

[vesse] Mark Vessella
Advanced Topics in Computer Graphics: Spherical Harmonic Lighting
http://web.cs.wpi.edu/~emmanuel/courses/cs563/S05/talks/mark_w5_spherical_harmonic_lighting.pdf

[patol03] Teemu Maki-Patola of the *Helsinki University of Technology*
Pre-computed Radiance Transfer
http://www.tml.tkk.fi/~tmakipat/prtf_fixed.pdf

[ronnb04] Simon Ronnberg of *Umea University*
Real-Time Rendering of Natural Illumination
http://www.cs.umu.se/education/examina/Rapporter/SimonRonnberg.pdf

[gautr05] Pascal Gautron, Jaroslav Kˇrivánek, Kadi Bouatouch, Sumanta Pattanaik
Radiance Cache Splatting: A GPU-Friendly Global Illumination Algorithm
http://www.irisa.fr/siames/Pascal.Gautron/projects/rcsplat/rcsplat.htm

[chapm04] Nick Chapman *Real-Time Radiosity*
http://homepages.paradise.net.nz/nickamy/realtimerad/realtimerad.html

[elias] Hugo Elias *Radiosity*
http://freespace.virgin.net/hugo.elias/radiosity/radiosity.htm

[vegda00] Nathan Vegdahl *Real-Time Radiosity*
http://www.gamedev.net/reference/programming/features/rtradiosity/
http://www.gamedev.net/reference/programming/features/rtradiosity2/

[jense01] Henrik Wann Jensen
Realistic Image Synthesis Using Photon Mapping
Published by A K Peters, Massachusetts, 2001

[larse04] Bend Dalgaard Larsen and Niels Jørgen Christensen
Simulating Photon Mapping for Real-time Applications
http://www.cs.sunysb.edu/~mueller/teaching/cse690/imm3192.pdf

12.Appendix A: Project Specification

12.1 Motivation & Aims

On the hardware of today, real-time graphical applications rely on basic illumination models such as Phong and Lambert. These models by themselves are unable to correctly process many lighting phenomena such as translucent objects, reflections, shadows and volume fog. There are also many cases in which these techniques and their extensions can cause undesired rendering artefacts. I will be researching models that solve one or more of these problems.

However, the aim of this project is to show that it is possible to combine most or all of these lighting phenomena into a single global illumination model. In this project, I will research and demonstrate various global illumination solutions for real-time applications, compare them, and then select one for further study.

12.2 Areas of Research

The main focus of research will include the most effective ways of reproducing accurate lighting in a real-time simulation. These include, but are not limited to:

- Static & Dynamic Light Mapping
- Spherical Harmonics
- Global Illumination using Wavelet theory
- "Instant Radiosity"
- Radiance Caching/Mapping

Also, this project will research ways of integrating these techniques into a real-time, interactive simulation. It is therefore essential that interactive speeds be maintained in all the prototypes where possible. This leads to:

- Implementing a global illumination solution on a GPU
- Optimising each model to run at interactive speeds
- Researching possible improvements for current or future (possibly hypothetical) hardware

12.3 Schedule

- ✓ **November 2005 to January 2006**

In this period, I aim to achieve the minimum and first boundaries of the criteria.

- ✓ **February 2006 to March 2006**

In this period, I aim to achieve everything in the each boundary.

12.4 Criteria

Table 2 - Overall Criteria

Overall Criteria	
Minimum	The following solutions are researched: • Light-mapping using ray-tracing • Light-mapping using photon-mapping • Instant Radiosity
Boundary 1	The following solutions are researched: • Irradiance Caching or Radiance Cache Splatting • Spherical Harmonics
Boundary 2	The following solutions are researched: • Irradiance Caching and Radiance Cache Splatting • Other solutions have been discovered and documented appropriately. At solution must have been prototyped.

After a suitable number of models have been researched, one will be selected for prototyping and evaluation.

The prototyped solution's documentation must describe:

- The applicability to current hardware
- The quality and speed on a high-end GPU
- How the model has been approximated or adapted to work on a GPU
- If the model is suitable for further evaluation.

Each prototype must:

- Run at a minimum of 25 frames per second (where possible)
- Be stable
- Clearly demonstrate how each model is superior or inferior to other models.

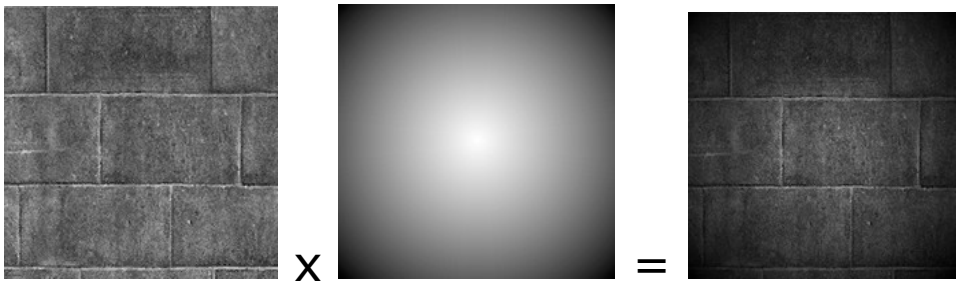
13.Appendix B: Current Solutions

Presented here is the research that I have performed on a selection of the current global illumination solutions that have been developed. Section 13.1 covers *Light Mapping*, which is a way to embed lighting data within a texture for display. Section 13.2 covers a method called *Pre-computed Radiance Transfer* which allows many kinds of phenomena to be simulated at real-time using pre-computed radiance information. Finally, there are radiance and irradiance - hence forth referred to as (ir)radiance – cache based methods covered in section 13.3.

13.1 Light Mapping

13.1.1 Light-Mapping Overview

Lightmapping is a technique that involves using pre-generated lighting data and applying the results to geometry in real-time. The lighting data for each face is stored as a texture that real-time applications can modulate with the full-bright scene to simulate lighting.



13.1.1.1How Are Light-Maps Generated?

The process behind lightmap generation is invariably a slow one. However, it does allow a good number of natural phenomena to be stored with the lighting data, allowing for very realistic results and yet maintaining a reasonable speed.

Lightmap generation can be done in a number of ways. Almost every technique involves taking each face in a 3D mesh, subdividing it into a series of *patches* and calculating global illumination at that point. This illumination data then gets

stored in a texel of a light map for that face. Each texel of this lightmap is called a *lumel*.

There are a number of problems that need to be solved before we have a complete light-mapping solution. The first is that we need to find an appropriately sized lightmap for each mesh. The second is that we need to subdivide each of our faces into patches. The third is to work out the illumination contribution for each patch (representing one lumel) on that face.

13.1.1.2Selecting a Light-Map Resolution

The first problem is relatively trivial for one face. However, each face is of an arbitrary size so despite being trivial to implement, it is very important that it is done correctly. Failure to include this step would result in having to use a uniformly sized lightmap for every face. Imagine a fixed size of 128x128: on small faces this would be a terrible waste, and on a very large face this would cause aliasing and frankly look quite terrible. Therefore it is important that we build an appropriately sized lightmap for every face.

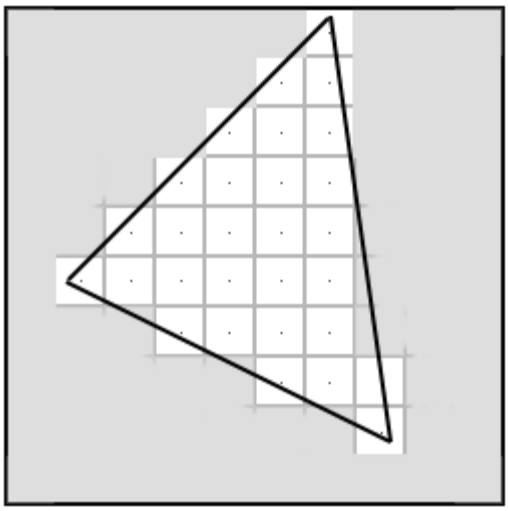
However, we are not only dealing with one face per lightmap. This would be not only wasteful use of storage and video memory, but also "seams" can appear between faces and most importantly it is not feasible to draw each face with its own texture. This turns a trivial process into a complex one, as each face of a model must be fit into a single lightmap in the most efficient manner and also this lightmap must wrap seamlessly around a mesh.

Because graphics hardware requires a texture size that is of a power of 2, our texture sizes are limited to 2, 4, 8, 16, 32, *et cetera*. A size of lightmap above 2048 is even possible but impractical, though this would obviously depend on the application. The simplest way of handling the lightmap size is to test to see if the face's dimensions exceed a given threshold and select an appropriate resolution based on this.

13.1.1.3Subdividing Our Face

Because we are dealing with textures, each one must be generated to surround an arbitrarily sized polygon. When we select a resolution we must divide the face into patches that correspond to the lumels of the lightmap. We can do this by

simply projecting the face into 2D, whether by a world-to-texture-space matrix or eliminating one axis, followed by dividing the dimensions of the polygon by the dimensions of the lightmap. Handily, this will also allow us to easily find the UV coordinates for each vertex.

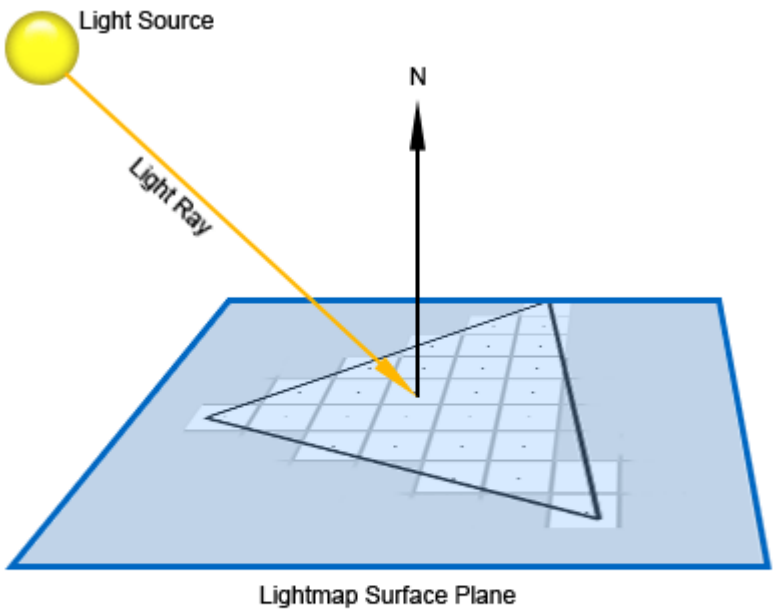


The face is divided up to match the lumels in the light map.

The first task is to find which lumels fall inside the area of the face. We find the centre point of each lumel and test if it is within the bounds of the triangle. If it falls outside the area, it is marked as invalid. If it falls *within* the area, we mark it as valid and find the illumination contribution for this lumel.

13.1.2 Finding the Illumination Contribution

As I mentioned above, when we find a valid lumel we light it using a global illumination model. Whichever model we are using, we must first find the world space position of this lumel so that lighting calculations can be easily solved. To do this, we can use a texture-to-world space matrix to transform our texture-space coordinate into the world. Alternatively, we can use the previously found UV coordinates to interpolate between the 3D points in world space. This point can be fed into our model to find the lighting contribution here. The lighting models used for this are beyond the scope of this article.



Once we have processed each lumel, we must then handle the invalid lumels. This involves filling the lumel with the value in its nearest *valid* neighbour. This is done because in cases of bi- or tri-linear filtering we may get a combination of valid and invalid data, which would produce black lines around the edges of our face.

13.1.3 How are Light-Maps Used in Rendering?

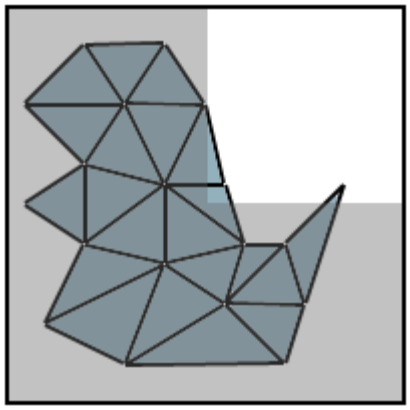
This is a relatively simple process and can be done one of two ways.

The first is using a multiple-pass method, where only the diffuse scene is rendered to the buffer. Then the scene is rendered again but only with the lighting data as the texture. These two images are modulated to give the final image.

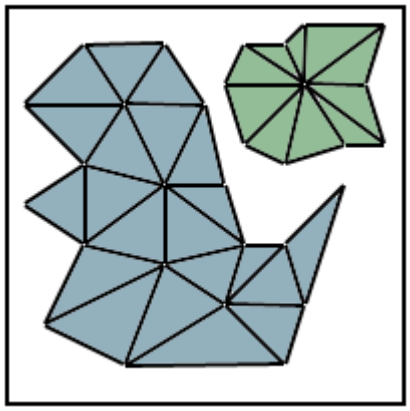
The other is more effective for real-time hardware. Using *multitexturing*, the diffuse and lighting passes can be rendered as one pass. When rendering, both textures are passed to the graphics hardware and are modulated together when the geometry is rendered.

13.1.4 Storing Light-maps

The storage of light-maps is unfortunately very expensive. A texture must be created for each surface and because each surface may not necessarily be square, we are wasting a lot of space, as shown here:



This can be overcome by *recycling* wasted space in this manner:



Here we make sure that our texture is optimally used by adding more geometry in to the remaining area.

13.1.5 How Light-Mapping is Useful

- **Fast rendering.**

  Lightmapping is perfect for current generation hardware as it is not only fast but produces reasonable results.
- **Broad Simulation.**

  Lightmapping can be made to support a wide variety of physical phenomena, including shadows and caustics.
- **Flexible use of Illumination Models.**

The lightmap generation process can be made to use any kind of global illumination algorithm to give more realistic lighting.

- **Simple.**

  Lightmapping is a simple technique and one that has been used for many years in real-time applications to great effect. It is relatively time consuming to generate and store lightmapping data, but once this process is done it rarely needs to be done again.

13.1.6 How Light-Mapping is Flawed

- **Static.** Lightmapping is only good when working with static geometry. It is not feasible to regenerate the lighting data every frame and so any geometry that is included in the lightmapping process must not move or the lighting data will become invalid.
- **Aliasing.** In order to keep asset sizes low and speeds high, lightmaps are usually generated at very low resolutions. This can produce aliasing in the lighting data when a lightmap is stretched over a polygon that is quite large.
- **Space Inefficient.** For realistic and high quality lighting, lightmaps should be at a reasonable resolution. Also, because one polygon is mapped to a square area, a lot of texture data is wasted on areas that are outside the range of the polygon.
- **Slow to Generate.** Lightmap data can be very expensive and time consuming to generate, though this does depend on the resolution and illumination model used to generate them.
- **Prone to Artefacts.** In many situations, artefacts can arise from mishandling of transparent geometry, aliasing or tearing between faces, bilinear filtering and a number of other problems. Most of these can be rectified, however.

13.2 Pre-computed Radiance Transfer (PRT)

13.2.1 What is Pre-computed Radiance Transfer?

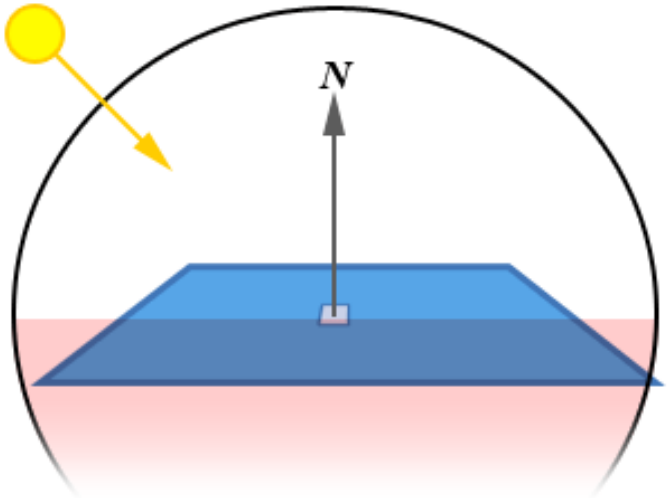
PRT (Pre-computed Radiance Transfer) involves the storage of radiance data developed by an external process and using it to light a scene in a real-time application. The radiance data is stored as a series of *transport coefficients* for each model and using these we can calculate the exit radiance for each pixel based on spherical harmonic illumination. Because of its pre-computed nature, this technique therefore also allows for dynamic self-shadowing, caustics and sub-surface scattering. One nice effect that is caused by PRT is soft shadows.

Spherical harmonics allows us to approximate the exit radiance for any point in 3D space with an arbitrary incoming light signal. This light can come from any number of sources, provided that the light they emit is of a low frequency. This therefore allows for dynamic lighting with soft shadows and colour bleeding to be simulated at real-time speeds.

Because real-time graphics applications generally deal with polygonal faces with an associated normal, this article concentrates on illumination using a hemisphere and ignores spherical harmonic lighting using whole spheres unless otherwise specified.

13.2.2 How Does Pre-computed Radiance Transfer Work?

Spherical harmonics and Pre-computed Radiance Transfer work on the principle of calculating illumination for a particular point in 3D space based on how much light it receives within a spherical or hemispherical domain. If we imagine this point as part of a plane being lit by one light source, we can imagine that this point is lit provided that the normal of the plane faces towards the light source. Therefore, the lighting data is the accumulation of each light source's contribution. Mathematically, it can be seen as the *integral of all light contributions over the hemisphere.* Due to the model's spherical nature, we can substitute or combine light sources with a spherical environment map.



The light at any point is given by the accumulation of light over a hemisphere in the direction of the point's normal.

It is obviously not feasible to integrate these values, so we use a stochastic approximation using a Monte Carlo Estimation system. We take many random samples over the hemisphere and average them and this represents an approximate result. However, even if we were to get this process running at an adequate speed, it would also mean that we would have to store lighting data or an environment map *for every vertex*. This is obviously totally unacceptable.

It is therefore more feasible to use a single function to calculate the radiance data at each point from a single environment. This is done by taking the incident radiance for each point and projecting it into the spherical harmonic basis. This will help us obtain our exit radiance for this point based on a single set of data rather than store data for each point.

Before answering the question, "How is the Radiance Data generated?" we must know how the spherical harmonic data is stored. For each object we must generate what are called *transfer vectors*. These are a list of coefficients that we modulate with our spherical harmonic bases functions at run time to reconstruct a low-frequency light signal. It is possible to generate the coefficients for every texture-space pixel on a 3D object, but it would be more efficient to generate these coefficients for every vertex.

At run-time, the illumination of a particular point can be found by finding the sum of all the dot products of each coefficient with the lighting data given by the spherical harmonic data. This is can be termed as *reconstructing* our light transport function (*f'*) from our coefficients, therefore allowing us to reconstruct the illumination output without having to perform the complex operations each frame for each point.

$$\tilde{f}(\mathrm{s}) = \sum_{i=0}^{n^2} c_i y_i(\mathrm{s})$$

Where:

l = output light contribution

y_i = spherical harmonic function[12]

c_i = coefficient from transfer vector

$f'(\mathrm{s})$ = the approximated light transport function for s

Note that *f'(s)* is an *approximation* of the original function, *f(s)*. The number of coefficients (n^2) in a transfer vector depends on the order*, n,* of spherical harmonic used. The higher the order of spherical harmonic, the closer *f'(s)* will be to *f(s)*, but at the expense of computation time. y_i is the spherical harmonic function that is calculated for the lit scene. This can be done by lighting a sphere and using the result as a spherical environment map to sample. This should obviously be done in the real-time application but not each frame if possible. It is generated using this equation[13]:

$$y_l^m(\theta,\varphi) = \begin{cases} \sqrt{2}K_l^m \cos(m\varphi) P_l^m(\cos\theta), & m > 0 \\ \sqrt{2}K_l^m \sin(-m\varphi) P_l^{-m}(\cos\theta), & m < 0 \\ K_l^0 P_l^0(\cos\theta), & m = 0 \end{cases}$$

l = polynomial band index

m = <look me up> ∈ [-l, l]

K = scaling parameter

This uses the *Associated Legendre Polynomials* theory (based on the orthogonal polynomials theory), which is represented by P(l, m), where l is a discrete value greater than zero representing the polynomial band index and m is traditionally a

[12] It is useful to store the spherical harmonic basis functions into a single vector. For this, we use:

$$Y_l^m(\theta,\phi) = Y_i(\theta,\phi)$$

Where $i = l(l+1)+m$.

[13] Taken from [green03]

This equation uses polar coordinate systems. To convert between Cartesian and polar, we use the equation: *(sinΘ, sinΘsinΦ, cosΘ) = (x, y, z)*

discrete value between zero and *l.* However, when working with real spherical harmonics instead of imaginary numbers, we assume a special case where *m* is in the range –*l* and *l*. See the end of this article for more on Associated Legendre Polynomials. For further information, see [green03].

The theory behind generating these coefficients is also lengthy, so to reduce clutter I will only present equations that are directly relevant to the real-time solution.

To calculate a coefficient for a given polynomial band we simply integrate the product of our light transport function, *f*(s), (whatever it may be) and the lighting coefficient, *y*. You will notice this takes a parameter, *s*, which represents a random sample.

$$c_l^m = \int f(s)\, y_l^m(s)\, ds$$

These coefficients can also be changed to handle glossy/shadowed surfaces by extending the transfer vector into a matrix. The elements of this matrix can be generated in a similar way to the vector coefficients:

$$\boldsymbol{M_{ij}} = \int_s b(s)\, y_i(s)\, y_j(s)\, ds$$

This outputs a matrix that can be stored with each vertex similar to the transport vector. This equation takes a parameter b(s) which is known as a *shadowing function,* which returns a real number in the domain [0, 1] which describes the amount of light reaching the surface at that point. This function could be evaluated using a ray-tracer or some other occlusion-testing technique.

13.2.3 Using the Pre-computed Data

It is now possible to use this pre-computed data in a real-time application. This is best achieved using shaders, allowing us to combine the PRT data on a programmable pipeline. Because it is best to be dealing in per-vertex PRT, we can use this to interpolate the radiance transfer results over a face. If we were to use per-texel PRT, we could do this solely in a pixel-shader.

First we would have to load in our coefficients from our model file. It is best to have this stored in a file for later recall rather than attempting to generate this data each time. Our lighting coefficients, *y*, can be either saved to a file or generated for each environment. We simply pass in our transfer vector and lighting coefficients into a shader and evaluate them using the function given above, given again for your convenience:

$$\tilde{f}(s) = \sum_{i=0}^{n^2} c_i y_i(s)$$

Obviously this could be optimised greatly. One way we could do this is to reduce the overhead created by sampling so many times for each vertex by compressing our PRT data. The algorithm recommended by *Microsoft* is *Clustered Principle Component Analysis* or CPCA. This reduces the number of coefficients in a transfer vector by compressing them. Refer to the MSDN library at [msdnb] for more information.

13.2.4 How Pre-computed Radiance Transfer is Useful

- **One Pass.**

  Lighting data can be generated in only one pass, unlike many other techniques. This is good because it helps us reduce the fill-rate.
- **Large Range of Phenomena at Low Cost**.

  This model can simulate a wider range of phenomena built into a single model, such as soft shadowing, caustics and radiosity (including interreflectance). These phenomena are recorded in the PRT data, so they come at no extra overhead penalty.

13.2.5 How Pre-computed Radiance Transfer is Flawed

- **Low Frequency Only.**

  This method involves reconstructing a frequency signal given a set of basis functions and coefficients in which to control them. If we were to deal with high-frequency lights, we would need many more basis functions and coefficients. This is also the reason why specular objects are not simulated well in PRT.
- **Rigid Body Only.**

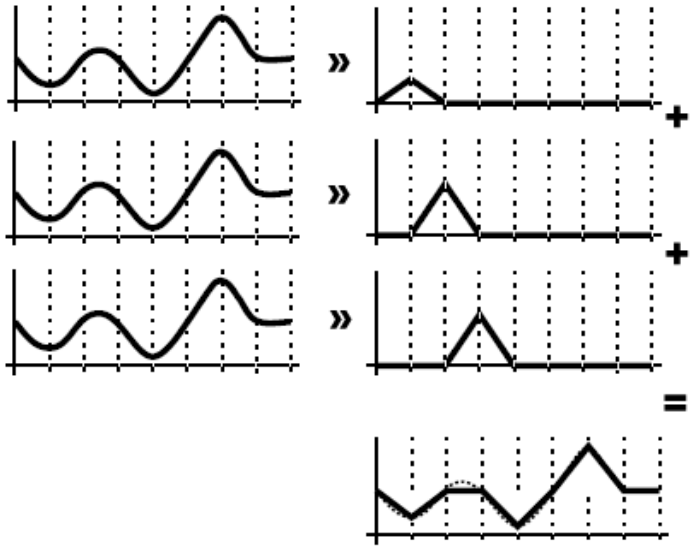
This method only applies to rigid bodies. If an object is dynamic in shape the PRT data becomes immediately invalid. It can handle affine transforms, however.

- **Only Works on One Object.**

  This can only apply to a single object because it is not possible to generate lighting data for dynamic objects.

13.2.6 Associated Legendre Polynomials

The *Associated Legendre Polynomials* theory is based on the *orthogonal polynomials* theory. This theory dictates that when integrating two orthogonal polynomial functions we get zero if they are different, or if they are the same we get some constant. Because we are using orthogonal basis functions, this constant is always one. This allows us to break down a signal into a series of *basis functions* which will allow us to approximate the original when they are added together[14].



Spherical harmonics utilises this theory in that we break down an incoming lighting signal into spherical harmonic basis *coefficients*: we can approximate the original lighting function by performing a dot product of the transfer vector with these coefficients.

[14] This is why we must use a low frequency signal when using spherical harmonic lighting: if we use a high frequency signal it would require an enormous amount of orthogonal basis functions to accurately reconstruct the signal.

I refer you back to this equation:

$$y_l^m(\theta,\varphi)=\begin{cases}\sqrt{2}K_l^m\cos(m\varphi)P_l^m(\cos\theta), & m>0\\ \sqrt{2}K_l^m\sin(-m\varphi)P_l^{-m}(\cos\theta), & m<0\\ K_l^0P_l^0(\cos\theta), & m=0\end{cases}$$

Where function *K*(), which is a scaling factor given by:

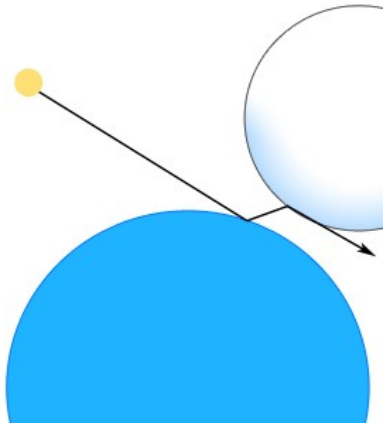
$$K_l^m=\sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}$$

The reference to the function *P*() here refers to the Associated Legendre Polynomial with a "band index" *l*. The band index ultimately determines the order of the spherical harmonic basis functions. The number of spherical harmonic coefficients generated is equal to *l(m*+1*)+m*. See the image on the following page for a visual representation of the band index, which refers to the first five spherical harmonic functions. This image is from Robin Green's "*Spherical Harmonic Lighting: the Gritty Details*".

13.3 (Ir)Radiance-Based Methods

13.3.1 What is (Ir)Radiance Caching?

In real life, light will bounce many times off surfaces before reaching the eye. This leads to what is referred to as "colour bleeding" between diffuse objects and also soft shadowing effects. In traditional CG techniques these bounces to not occur: the light is simply calculated directly from a point in space and this is what is written to the image (hence the term *direct illumination*). Because of this direct lighting approach we lose this effect, so in order to generate realistic images we must somehow approximate the continual reflection of light rays.



Because we cannot feasibly compute the continual reflection of light rays in real-time, methods have been developed that simplify the radiosity calculation so that we can approximate realistic colour bleeding between objects in an interactive application. These methods rely on what we call radiance and irradiance caching; these are methods for recording which areas of an image require more lighting samples to be taken due to mesh and lighting complexity. These samples allow us to calculate how much light each object contributes to its neighbours. Due to this sampling, however, this method cannot efficiently handle high-frequency light phenomena such as hard-edged shadows/lights or caustics.

13.3.2 Traditional Radiance Cache

Traditional radiance caching records how much light is being received at a particular patch in space. This is done by tessellating a 3D mesh into very small patches and computing the radiance at this patch by rendering the scene from that patch's point of view into a hemi-cube. The radiance at that point is then the integral of all the lighting values in that hemi-cube. This can also be done using a high-order spherical harmonic. However, subdividing a mesh into such a high poly model and rendering the scene thousands of times is very expensive and not desirable.

However, it is still achievable if we take into account the model complexity. For example, a large, flat surface will require many less samples to compute an accurate approximation as these samples can simply be interpolated across the face, yet a curved mesh will require many more samples to simulate accurately.

Another disadvantage is the way that these values are stored for use. Radiance or irradiance values are stored in a huge tree for 3-dimensional lookups when a

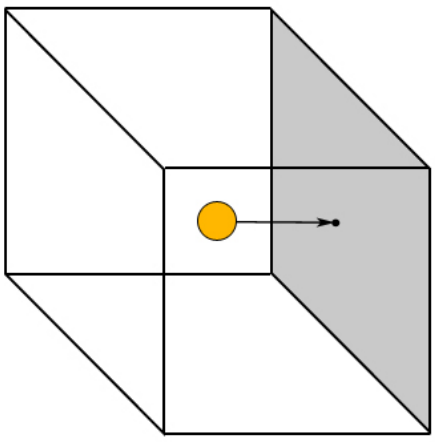
point is sampled. Due to limitations on the GPU, this process cannot be implemented effectively, if at all. We must find methods of finding the radiance or irradiance of a surface without the need for this complex data structure.

This method is unbearably slow and not worth considering for real-time applications. However, the reason I that I have introduced this method is because it can be significantly modified to work on real time hardware. I now propose various methods of simulating radiosity for interactive applications.

13.3.3 Radiance Mapping

This method was devised by Nathan Vegdahl (vegda00) who posted an article to *GameDev.net* in 2000. This method involves using the vertex normal for an object to sample radiosity data from a cube-map. This cube-map is a low-resolution representation of the entire scene from the origin of the object in question. Because we wish to simulate radiosity, not reflections, this cube map is blurred so that each pixel contributes evenly to every surrounding pixel within half of the cube-map's size. This allows for a smooth change in incident radiance.

Once the incident radiance has been obtained, this value is added to the lighting contribution which can be determined using traditional Lambertian or Phong methods.



An example of sampling from a cube map.

However, this will not produce proper results. Colour bleeding is a phenomenon caused by the scattering of light in all directions when contacting a diffuse surface, but this technique currently only samples one direction. To improve accuracy, we can perturb the normal of the face using a stochastic technique and sample the cube-map many times for each point. We wish to sample the cube-map in 180 degrees in a semi-sphere, so we use a Monte Carlo technique to generate the perturbed normals to sample with. The radiance is then the average of these radiance values.

This repeated sampling can produce overhead, so it is perfectly possible to perform this operation per-vertex instead of per-pixel and use Gouraud shading to interpolate these radiosity values over the surface.

This technique is advantageous because it is perfectly suited to graphics hardware, but is flawed in that each object must be very small and must have a cube-map rendered each object, each frame. However, the cube map only needs to be very small (32x32x32 or even 16x16x16) to produce decent results. Another bonus is that objects further away already contribute less to the radiance of the object because of the cube-map view. This means that we do not have to attenuate our radiosity values.

13.3.4 Instant Radiosity

This method relies on tracing random rays from a light source using Monte Carlo methods and placing a point light source where this ray hits an object. The ray is then either killed or bounces in a random direction based on a random probability. If the ray bounces, the light factor of the ray is attenuated so that each bounce leaves a dimmer light source behind. This method is slow as it involves the computation of a high level of point light sources, but can give excellent results for very simple scenes. This method is extremely simple but has little use on today's hardware, and is not worth studying further.

13.3.5 Irradiance Cache Splatting

Irradiance Cache Splatting (RCS) is probably the most desirable of the three methods presented here. Developed by Pascal Gautron *et al* (gautr05), this is a reasonably fast[15] method based on "splatting" first-bounce radiance data to an

[15] It is faster in comparison to current implementations of real-time irradiance cache.

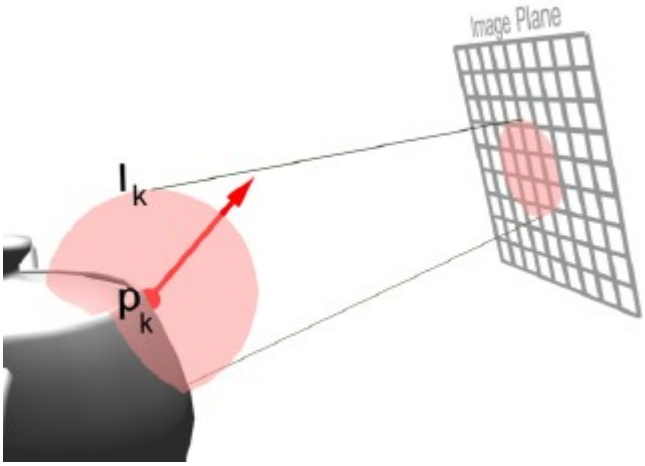
off-screen buffer for combining in the final pass. "Splatting" involves rasterising a circular area to a buffer that represents lighting data. It takes full advantage of the GPU's programmable pipeline and can also be adapted to handle glossy (but not highly specular) objects.

This method works on the reversed principle of the standard radiance cache. In the case of standard radiance cache the algorithm determines which records[16] contribute radiance data for each pixel and how much they contribute by. These values are then accumulated and written to the image plane. Radiance cache splatting does the opposite in that it "splats" each record onto a *splat buffer* immediately, accumulating the weights and radiance data in real-time. The use of these weights ensures that no splat value contributes to values outside of its sphere of influence. The radiance cache splatting algorithm works by only selecting records that are guaranteed to contribute for a given pixel and this contribution is added to the splat buffer.

The splat buffer is the heart of radiance cache splatting. This stores the accumulated radiance data and the weights of this data in a 2-dimensional buffer of the same dimensions as the image plane. When writing the fragments from the scene (which is lit and shadowed normally), this fragment is combined with the existing splat data using a pixel-shader to produce the end result. This combination process involves dividing the accumulated radiance data for that pixel by the accumulated weights.

This process works by computing the radiance for each pixel of the image and rendering a circle of user-defined dimensions to the splat buffer, which represents the lighting data at that point in space:

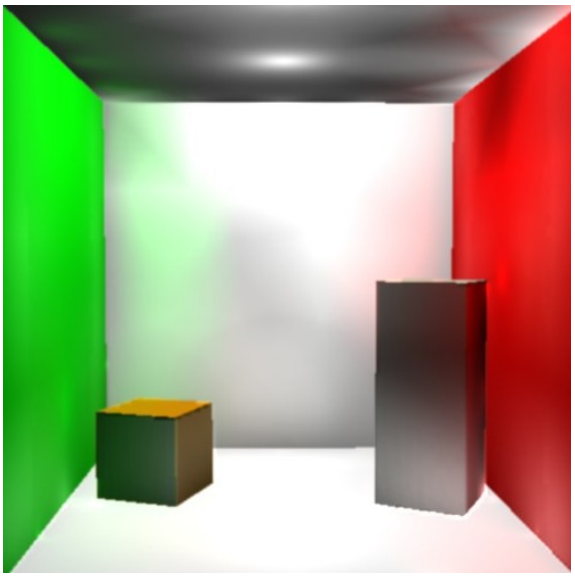
[16] A record refers to an irradiance record. In traditional radiance caching the world is filled with these records which contain all the information for lighting the point at which the record is held in 3D space.



Splatting to the buffer. Image from [gautr05].

This means that not only can we take advantage of the GPU's shape rasterisation, but we can also quite nicely use the render-to-texture capabilities. To do this, we select irradiance records from the scene that lie "under" the pixel we are calculating lighting values for and add the record's contribution to the splat buffer in the shape of a circle. We also record the cumulated contribution amount at this pixel, called *w*.

However, the disadvantage to the method is that we must pass back the splat buffer to CPU memory after the first phase of the splatting is finished. The reason for this is that we must make sure that the entire image has an evenly distributed accumulation of splat records. Failing to ensure this is so will result in patchy results.



A patchy image resulting from generating too few irradiance records.

To do this, we maintain a user-defined level of accuracy, which has been named *a*. Each pixel of the splat buffer must conform to *a*, so we must make sure each pixel has the appropriate number of contributions and if this is not so ($w < a$), we must splat a new radiance record to the buffer for that pixel. The buffer must then be passed back to the graphics hardware. This process can stall the rendering pipeline, resulting in slow-down.

Despite this, this method is advantageous as it does not depend on tessellating the model to simulate the light data and it can be done per-vertex without losing too much accuracy. This method is also still quite accurate despite this difference to traditional radiance cache techniques. Finally, splatted data can be re-used each frame so animation and real-time interactivity is possible.

13.3.6 How (Ir)Radiance Caching is Flawed

The methods presented here have their own advantages and disadvantages which I have covered in the relevant sections, but (ir)radiance caching can suffer from common problems.

- **Few Phenomena Simulated.**
  (Ir)radiance caching does not simulate many lighting phenomenon such as shadows or caustics. However, this technique can be combined with regular techniques for simulating these effects if needed.
- **Low Frequency/Soft Lights Only.**
  Because we are relying on samples to generate our lighting data, it is inefficient to simulate high-frequency or sharp lighting phenomena as they would require a very high number of samples, far too many to be feasible on hardware.
- **Bad Handling of Point Lights.**
  Point lights are not simulated very well as they require an entire 360 degrees of computation to be performed. This method works well for simple, directional light sources such as spotlights.

14.Appendix C: Photon Splatting

14.1 Introduction to Photon Splatting

Irradiance Cache Splatting, as previously mentioned, is based on the *reversed* principle of irradiance cache. This means that instead of searching for nearby irradiance records for each pixel and determining how much each record contributes, irradiance records are instead "splatted" to a buffer to determine which pixels they contribute to. This eliminates the need for expensive complex data structure look-ups, such as a *k*d-tree.

To speed up the process further, however, we would require a simplified global illumination model. Irradiance Cache requires mathematics to compensate for distance attenuation, angular attenuation and other factors which is unfeasible on the current Shader Model 2.0 cards. In order to minimise the number of calculations, an approach based on Jensen's *Photon Mapping* (jense01) was selected. Photon mapping is ideal because of its simplicity: it does not require much complicated mathematics to achieve a pleasing result and it is a very scaleable solution. Jensen's original method involves an expensive *k*d-tree look-up but because of the splatting process this look-up is removed.

When developing this algorithm, a benchmark machine was used: 2GHz CPU, 1GB of RAM and an nVidia GeForce 6600 running Windows® XP. This is a common system and therefore the implementation was forced to meet the requirements of running on current hardware.

14.2 Photon Computation

The records – or more accurately *photons* in this case – can be determined any number of ways. In this implementation each vertex had its own record meaning that scenes must be highly tessellated in order to obtain accurate results. However, other methods can and should be employed where possible. For example, the *Irradiance Cache Splatting* algorithm used an expensive read-back of the textures to make sure that they maintained accuracy over the entire image, but this is not feasible without PCI-Express hardware.

A possible method is that photons could be "projected" into the screen. By selecting an even distribution over the surface of the image plane, photons could be added at uniform intervals over the resultant image. This is geometry independent and an efficient way of calculating photon data, but it may lead to inaccuracies towards the edges of the screen where photons are clipped. Also, because of the uniform spread, some diffuse inter-reflections may be missed if the spread is not large enough.

14.3 The Splatting Algorithm

The algorithm is a simple one. It can be easily managed on current graphics hardware and has the potential to be very fast or very accurate, depending on how the records are spread.

For each record, a circle is rendered to the splat buffer using the rasterisation engine and screen-aligned quads. For each pixel in this quad, the first-bounce lighting is calculated for that record (which will be in the centre of the quad). This incoming radiance is added to any values that are already in the splat buffer and the alpha channel is used as a weight.

At the final compositing stage, each pixel of the splat buffer is merged with the lit scene. Each splat buffer pixel is normalised (by dividing through by the accumulated weight) to maintain an even spread of photon data over the image. It is then added to the lit scene.

Here is this process in pseudo-code:

Listing 6

```
For Each Record k = { Pk, Nk, Ek, Rk }
Do
            Calculate Quad For Point Pk with Radius Rk
            For Each Pixel f = { Pf, Nf, Tf } in Quad
            Do
                  // Calculate the distance from the centre
                  d = |( Tf * 2 – 1 )|

                  // Ignore any pixels outside of this radius
                  if ( d > Rk ) Continue

                  // Calculate the distance from the fragment
                  // position and the record position
                  D = Pf – Pk
                  R = |D|
```

```
                    // Calculate the attenuation of the pixel
                    w = [Attenuation for this Pixel]

                    // Accumulate the attenuated incoming radiance
                    S(x,y).rgb += E_k * w

                    // Accumulate the weight
                    S(x,y).a ++
                End
End
```

Where k is the record currently being processed with position $\mathbf{P}_k$, direction $\mathbf{N}_k$, outgoing radiance $\mathbf{E}_k$ and radius $\mathbf{R}_k$ and where f is the fragment in the quad with position $\mathbf{P}_f$, normal $\mathbf{N}_f$ and texture coordinate $\mathbf{T}_f$.

The weight, w, is defined by calculating four things: the angle of incoming radiance, the angle of outgoing radiance, the distance attenuation and the splat attenuation given by d. This is given here:

Listing 7

```
// Calculate the angle of incoming radiance
i = N_f · −D

// Calculate the angle of outgoing radiance
o = N_k · D

// Calculate the distance attenuation
A = 1 − ( R / R_k )

// Calculate the final attenuation using inverse square
// and the circular fall-off, d
w = i * o * A^2 * ( 1 − d )
```

This gives us smooth records which are pleasing to the eye. The $(1 - d)$ term actually makes invalidates the model but was added to ensure that all photons fade off gradually. If this is not done, our photons will have an ugly sharp curve around the edges.

14.4 The Implementation

14.4.1 Implementation Overview

The implementation of the above algorithm was done using C++ in Microsoft's Direct3D and the algorithm itself was done completely on the hardware using Direct3D effect files (and Listing 9 – rcs_combine.fx). In order to ensure

interactivity and adequately demonstrate the capabilities of the algorithm, the application has a facility to rotate three coloured spheres around a central white sphere. A light is placed above the spheres, casting a white light onto the tops of each sphere. The algorithm shows that the incoming light for the tops of the lower spheres is being diffusely reflected on to the bottom of the central sphere.

The algorithm requires per-pixel information regarding the position, the normal and the illumination data about each pixel. The simplest way of doing this was to use off-screen buffers. Each of the used buffers was the same size as the back buffer. The positions (Image 4, page 81) and normals (Image 5, page 81) are stored in a buffer that is sampled in the pixel shader (Listing 3 – rcs_splat.fx). The lighting data is the result of rendering the scene with normal real-time lighting techniques as shown by Image 7 on page 83. This, for realistic images, could include the Phong or Blinn illumination model combined with a shadowing algorithm, such as shadow mapping.

The splat buffer is combined with a scene lit with a Lambertian lighting model in this implementation. However, it must be noted that the lighting colour contribution for that pixel when splatting was taken from this pass when calculating the outgoing radiance for that point. This means that only contributions facing in the negative z-axis are correct, but it is merely for demonstrational purposes. A more correct way of simulating the outgoing light would be to sample from a hemi- cube or sphere map of the scene visible from that record.

The implementation uses highly tessellated scenes in order to generate the records. The tessellation process is handled externally in 3D Studio Max and exported, which allows more control over the accuracy of the lighting data in areas of the model. A possible solution to this was covered in section 14.1 on page 72.

14.4.2 Speed and Efficiency

The model needed to be significantly optimised in order to run smoothly on the benchmark system. The above algorithm is the bare-bones of the photon display process and a lot has been approximated to obtain a greater speed. However, it

is possible that the algorithm above could be simplified further but at the expense of image quality.

Currently the simple demonstration scene (Image 1, page 79) provided with this report runs at approximately 25 frames per second on the benchmark system. However, the Cornell box scene (Image 3, page 80) which is highly tessellated to about 40,000 vertices runs at a mere 2 frames per second but is of the quality of a low resolution photon map. This demonstrates that even with extreme approximations, the method is still not suitable for large scale applications such as computer games.

There are measures to increase the performance of the current system. The translucent nature of the splat buffer increases overhead, but we can compensate for this by decreasing the resolution of the splat buffer. The sample size is then much smaller and large pixels will bleed over the edges of objects, but the mip-map and texture filtering support of graphics hardware helps reduce this effect. The splat buffer for the current implementation was a mere 320x240 in size, but still produced adequate results.

14.5 Results and Conclusions

This algorithm is currently in a very early stage of refinement. So far it has proved capable of handling diffuse inter-reflections at real-time speeds, but it is currently limited in what it can achieve because of the speed of graphics hardware today. It may be possible to optimise the rendering process and remove the need for so many passes using the next generation of shader model or GPUs that emerge on to the market.

Other than general efficiency issues, some of the methods used to generate the lighting data are still crude. The method implemented here is a very inaccurate method and has many drawbacks, but rendering the scene from the point of view of each vertex would be completely unfeasible on the benchmark hardware.

Despite its drawbacks, however, I do believe that this algorithm is worthy of merit merely because it can and *has* proved to do what it was intended. The Cornell

Box (Image 3 – Cornell Box, page 80) scene shows promising results, comparable to that of a low resolution photon map generated by an offline renderer.

Perhaps one day it can be extended to handle diffuse inter-reflections using variable radii, or use a more accurate sampling method to capture illumination data.

15.Appendix D: Image Output

15.1 Sphere Demonstration

Note: the diffuse inter-reflections here have been deliberately brightened for demonstrational purposes.

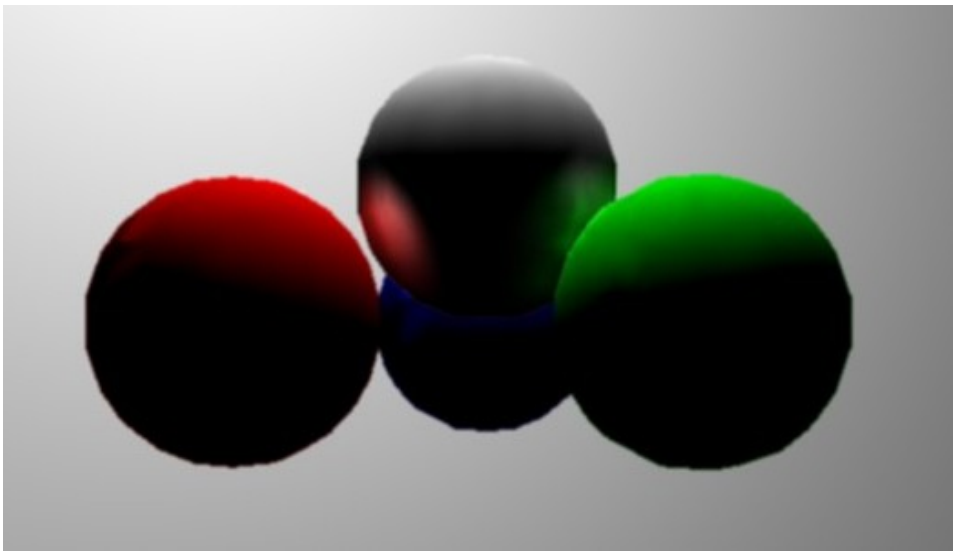


Image 1 – Sphere Scene 1

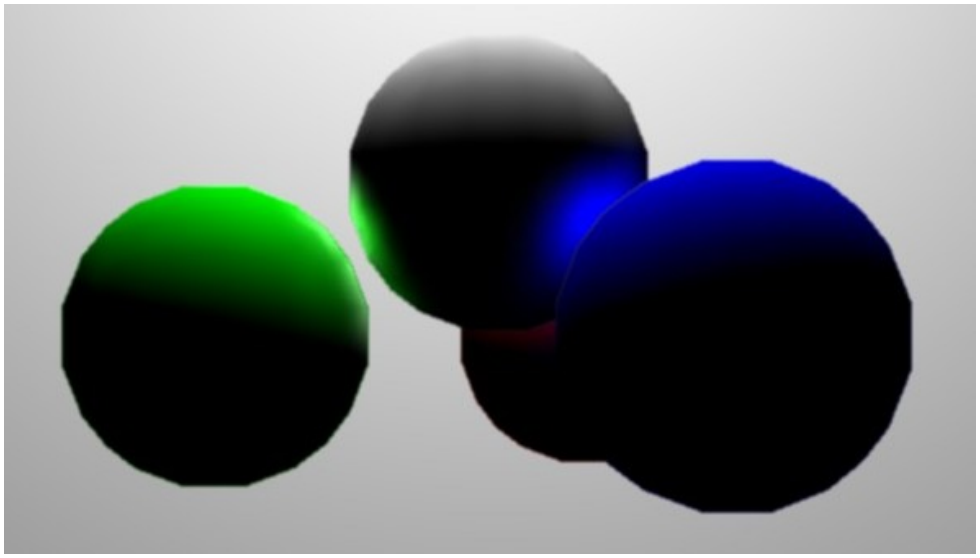


Image 2 – Sphere Scene 2

This scene consists of four highly tessellated spheres and a white box. This scene is approximately 18,000 vertices and runs at 25fps on the benchmark system.

15.2 Cornell Box Scene

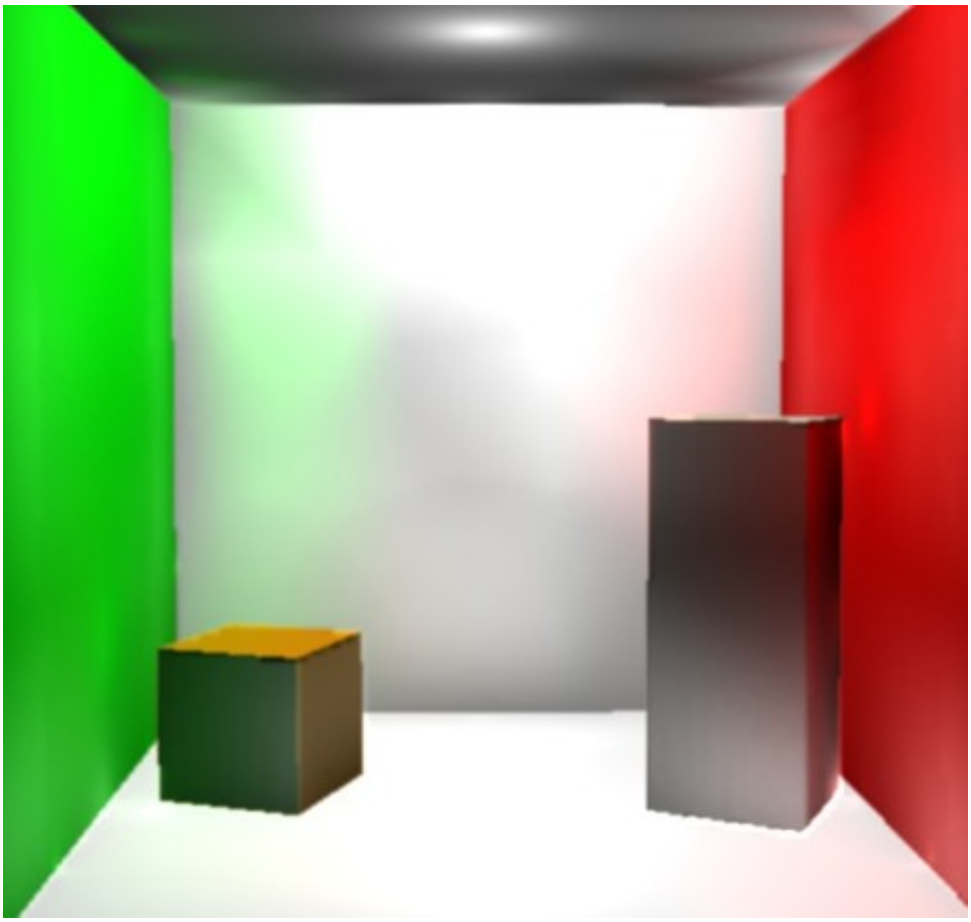


Image 3 – Cornell Box

This scene consists of approximately 40,000 vertices and ran at 2fps on the benchmark machine.

15.3 Off-Screen Passes



Image 4 – The Position Buffer

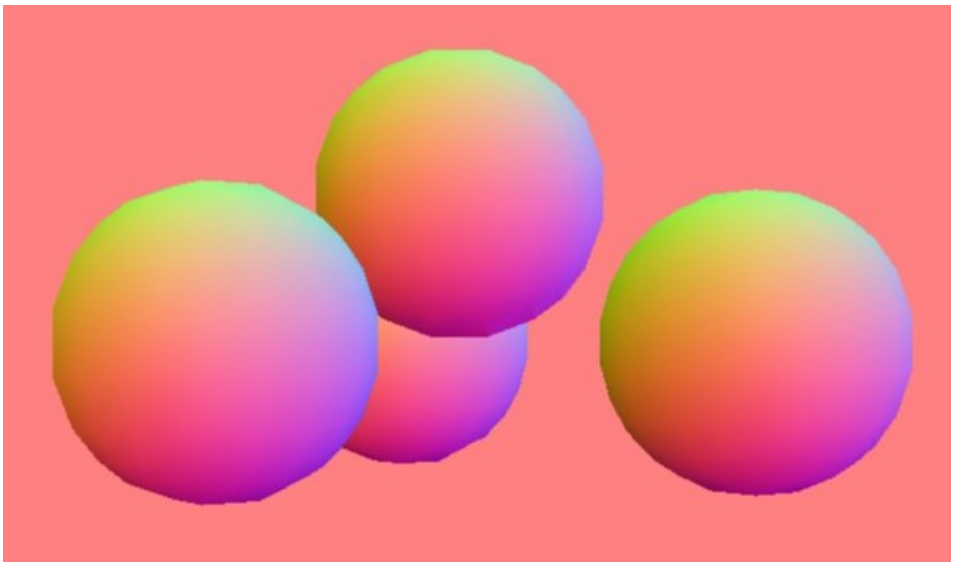


Image 5 – The Normal Buffer

This is a visual representation of the position and normal buffers.

15.4 The Splat Buffer

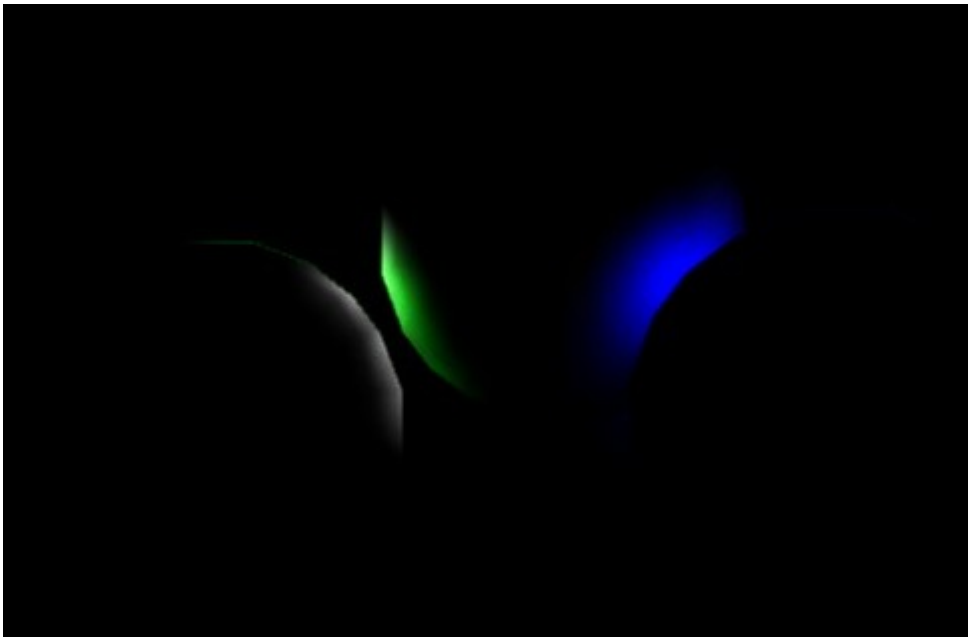


Image 6 – The Splat Buffer

This is an example of the splat buffer.

15.5 The Scene without Diffuse Inter-reflections

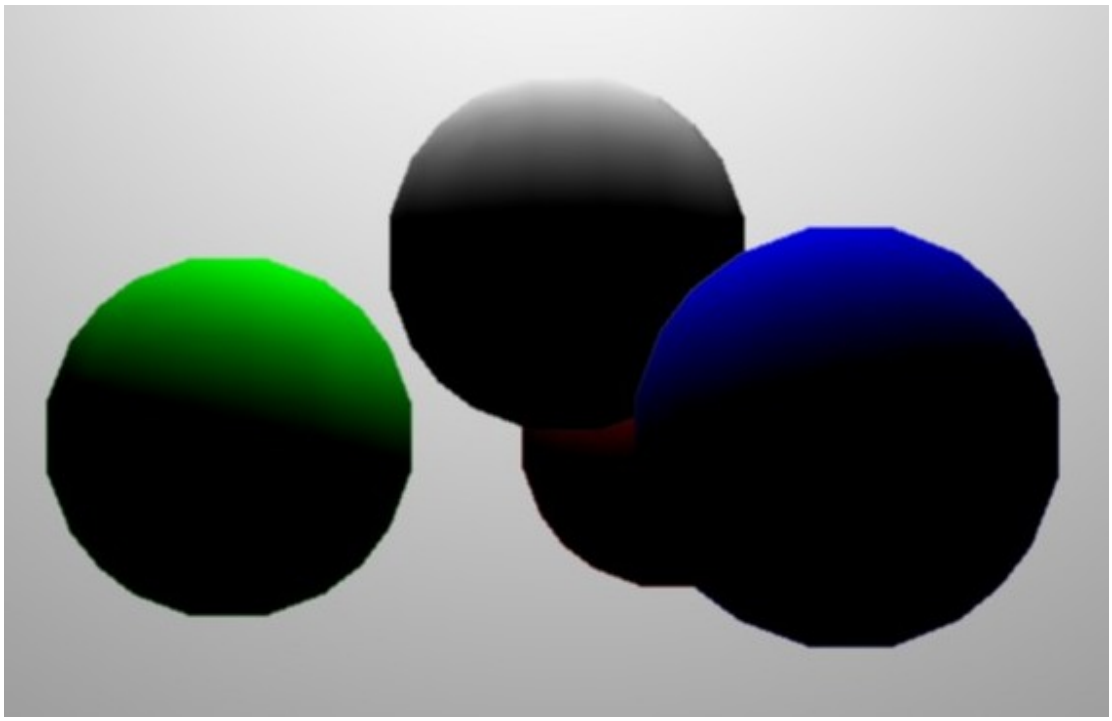


Image 7 – Normal Lambertian Lighting

This is the sphere scene again but without the diffuse inter-reflections.

16.Appendix E: Shader Listings

16.1 The Splatting Algorithm

Listing 8 – rcs_splat.fx

```
/*
       This is the radiance cache splatting algorithm.

       Make sure that you have blending enabled in ADD mode, as this will
       accumulate on the splat buffer.

       fAccuracy is the level of accuracy required.
       fRadius is the radius of the splat.
       fOneOverRadius is one over the radius.
*/

// Matrices
float4x4 mWorld;
float4x4 mWorldInv;
float4x4 mView;
float4x4 mViewInv;
float4x4 mProjection;
float4x4 mWorldView;
float4x4 mWorldViewProjection;

// Constants
float fAccuracy;
float fRadius;
float fOneOverRadius;

float3 vVertexNormal;
float4 vRecordPosition;

// The textures
texture  map_mask;
texture  Normals;
texture  Positions;
texture  Light;

// The sampler for the texture
sampler MaskSampler = sampler_state
{
       Texture = <map_mask>;
       MinFilter = POINT;
       MagFilter = POINT;
};
sampler NormalsSampler = sampler_state
{
       Texture = <Normals>;
       MinFilter = POINT;
       MagFilter = POINT;
};
sampler PositionsSampler = sampler_state
{
       Texture = <Positions>;
```

```
	MinFilter = POINT;
	MagFilter = POINT;
};
sampler LightSampler = sampler_state
{
	Texture = <Light>;
	MinFilter = POINT;
	MagFilter = POINT;
};

// Vertex shader input
struct VS_INPUT
{
	float4 Position : POSITION;
	float3 Normal : NORMAL;
	float2 UV0 : TEXCOORD0;
};

// Vertex shader output
struct VS_OUTPUT
{
	float4 Position : POSITION;
	float2 UV : TEXCOORD0;
	float2 Sc : TEXCOORD1;
	float2 Sc2: TEXCOORD2;
};

// Basic vertex shader
VS_OUTPUT VertexShader( VS_INPUT In )
{
	VS_OUTPUT Out;

	// Get the record position
	Out.UV = float2( In.UV0.x, 1 - In.UV0.y );

	// Get the input position
	float4 Pos = In.Position;

	// Move it into world space
	Pos = mul( Pos, mWorld );

	// Move from world into view and then projection space
	Pos = mul( Pos, mView );
	Out.Position = mul( Pos, mProjection );

	// Get the screen-space texture coordinates
	Out.Sc = Out.Position.xy / Out.Position.w;
	Out.Sc.y = -Out.Sc.y;

	// Get the screen-space texture coordinates for the record
	Pos = vRecordPosition;
	Pos = mul( Pos, mView );
	Pos = mul( Pos, mProjection );
	Out.Sc2 = Pos.xy / Pos.w;
	Out.Sc2.y = -Out.Sc2.y;

	return Out;
}

// Basic pixel shader
float4 SplatIrradiance( VS_OUTPUT In ) : COLOR
{
```

```
	float4 Out;

	// Get the screen texture coordinate.
	float2 ScreenPos = In.Sc * 0.5 + 0.5;
	float2 ScreenPosR = In.Sc2 * 0.5 + 0.5;

	Out = float4( 0, 0, 0, 0 );

	// Get the normal and position of the existing fragment
	float3 vNormal = tex2D( NormalsSampler, ScreenPos ) * 2 - 1;
	float3 vPosition = tex2D( PositionsSampler, ScreenPos )*2-1;
	float3 vColour = tex2D( LightSampler, ScreenPosR );

	// Get the 2D distance from the centre of the quad and this
	// fragment's position
	float2 UV = In.UV * 2 - 1;

	// Get the distance between the record and the
	// existing fragment.
	float3 vDistance = vPosition - vRecordPosition;
	float  R = length( vDistance );

	// Normalise the distance and find the dot product with
	// the normal.
	vDistance = normalize( vDistance );

	float  D = dot( UV, UV );

	if ( R <= fRadius &&
	       D <= 1 )
	{
		//float atten = 1 / ( R + 1 );
		float atten = 1 - ( R * fOneOverRadius );

		// This is the angle between the FRAGMENT normal and
		// the direction.
		// The normal must point in the OPPOSITE direction of
		// the direction vector.
		float angle_src = max( 0, dot(vNormal, -vDistance) );

		// This is the angle between the RECORD and the
		// direction.
		// The normal must point in the SAME direction of the
		// direction vector.
		float angle_dst = max(0,dot(vVertexNormal,vDistance));

		// Multiply the two attenuations.
		float angle = angle_src * angle_dst;

		// Invert the distance from the origin
		D = 1 - D;

		// Get the colour
		Out.rgb = saturate( angle*atten*atten ) * vColour * D;
		Out.a = 0.001;
	}


	return Out;
}

// Normal diffuse rendering
```

```
technique RenderModel
{
	pass P0
	{
		VertexShader = compile vs_1_1 VertexShader();
		PixelShader = compile ps_2_0 SplatIrradiance();
	}
}
```

16.2 Splat Buffer Combiner

Listing 9 – rcs_combine.fx

```
/*
	Basic textured shader.  Takes texture and diffuse colour and
	combines them.  Combines the vertices with the world and view
	matrices.
*/

texture  Texture;
texture  SplatBuf;

// The sampler for the texture
sampler TextureSampler = sampler_state
{
	Texture = <Texture>;
	MinFilter = LINEAR;
	MagFilter = LINEAR;
};
sampler SplatSampler = sampler_state
{
	Texture = <SplatBuf>;
	MinFilter = LINEAR;
	MagFilter = LINEAR;
};

struct VS_OUTPUT
{
	float4 Position : POSITION;
	float2 UV : TEXCOORD0;
};

struct PS_OUTPUT
{
	float4 Color : COLOR0;
};

// Basic vertex shader
VS_OUTPUT ScreenVertexShader( float4 Pos : POSITION,
                                        float2 UV : TEXCOORD0,
                                        float4 Diffuse : COLOR0 )
{
```

```
	VS_OUTPUT Out;

	// Transform the position
	Out.Position = Pos;

	// Set up the outputs.
	Out.UV = UV;

	return Out;
}

PS_OUTPUT SplatCombiner( VS_OUTPUT In )
{
	PS_OUTPUT Out;

	float4 Splat = tex2D( SplatSampler, In.UV );
	float Count = Splat.a * 100;
	if ( Count > 1 )
	{
		Count = saturate( 1 / Splat.a );
		Splat.rgb *= Count;
		Splat.a = 1;
	}
	else
		Splat = float4( 0, 0, 0, 0 );

	// Combine the texture and diffuse
	Out.Color = tex2D( TextureSampler, In.UV ) + Splat;

	return Out;
}

// This simply renders the scene to the target
technique RenderSplatScene
{
	pass P0
	{
		VertexShader = compile vs_1_1 ScreenVertexShader();
		PixelShader = compile ps_2_0 SplatCombiner();
	}
}
```

17.Appendix F: Document Statistics

Document completed on the 2nd of April, 2006.

Written by Peter J. B. Lewis (C3115495), University of Teesside.

Word count:

The word count statistics do not include this appendix or any footnotes.

Main document: 11002 words in 37 pages.

Appendices: 7,769 words in 38 pages.

Total: 0 words in 89 pages.