KING'S College LONDON

University of London

The implementation of a modern, GPU-accelerated radiosity renderer with OpenGL 4.5

MSci Computer Science - 6CCS3PRJ Final Project Report

(published version)

Author/copyright: Tamás Körmendi

Supervisor: Dr Richard Overill

April 13, 2018

Abstract

Abstract: This paper describes an in-depth way of implementing a modern, GPU-accelerated radiosity renderer, using OpenGL 4.5. First it briefly touches upon the history of rendering images with computers, especially concerning shading, shadowing and different global illumination techniques. It gives an overview and a more detailed history of the radiosity global illumination technique and the different ways to implement it. From this point the progressive radiosity technique is considered. The report also makes an argument for a modern implementation that should rely more on the GPU than the CPU. The implementation section first concerns itself with the design and requirements of the program, which also encompasses the libraries/technology stack of the application. A small frontend for the application was developed which shall be described at this point. In order to make the radiosity implementation part easier to understand the report also gives a primer on the purpose of GPUs, how they differ from CPUs, how the OpenGL 4.5 pipeline works and highlights and describes the more advanced GPU techniques used in the radiosity implementation. The main body of the implementation part presents two strategies for GPU-based radiosity: first an easier, naïve CPU-GPU hybrid one, followed by an advanced, GPU-only strategy. Detailed implementation notes are given for both strategies, each of them followed by an evaluation, results, conclusions and limitations section, with performance notes and screenshots given wherever appropriate. The report concludes with a final conclusions sections but before that – since unfortunately no software project is ever perfect – lays out the problems with the advanced radiosity implementation presented. In order to aid future work, possible fixes and plans are given for the problems laid out in the future work section. Additionally, three appendices are included. Appendix A includes a technical tutorial on visibility calculation with a hemicube, Appendix B is a short user guide and Appendix C is a commented guide on the structure of the program and where each of the described parts of the implementation can be found.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Tamás Körmendi

April 13, 2018

Acknowledgements

I would like to thank my parents and the rest of my family for their unending support throughout the project, even in my most difficult moments. I would also like to thank my supervisor, Dr Richard Overill, for introducing me to the wonderful world of computer graphics, for his quick responses and for his useful suggestions.

I dedicate this project to the memory of my cat, Turbó (08.08.2014 - 24.01.2018). May you raise hell in the heavens now.

Contents

Chapter 1

Introduction

The aim of this project is to implement a 3D renderer capable of simulating global illumination with radiosity. Radiosity is one of the oldest global illumination methods (first published in 1984 [1]) yet as opposed to other well-known global illumination algorithms (for example: ray tracing or path tracing) details for a modern radiosity renderer implementation with Graphics Processing Unit (GPU) acceleration are scarce. This report presents one way of accomplishing that task. It also provides a primer on GPU programming, since it is dissimilar to conventional CPU programming.

1.1 Report Structure

The report is presented in 3 main parts:

Chapters 2 and 3 are the background: they examine the history of computer generated images, provide some information about global illumination algorithms and go deeper into what radiosity is and the different radiosity techniques. Here the motivation for this project is mentioned along with outlining the direction the implementation chapters take.

Chapters 4, 5 and 6 are the implementation: the design and specification are described, followed by a whole chapter about GPU programming, since some familiarity with GPUs is required in order to understand the further parts. Chapter 6 presents two detailed radiosity implementations, along with a results/evaluation section for both.

Chapters 7 and 8 are the concluding chapters: Limitations of the current implementation and future work are set out and finally, a short summary of the report is presented as the conclusion.

Chapter 2

Background

2.1 Glossary

In order to make the background more understandable, here are the definitions of some of the non-obvious but commonly used terms:

- Shading: The act of determining how much light reaches the face of an object (for example, these faces can be the sides of a cube) from a light source and how much of it is reflected, thus giving the colour and shape of an object.
- Shadowing: The act of determining if a point in space is visible from a given light source. If the point is visible, it is not in shadow. If the point is not visible then it is in shadow. If only shading is used without shadowing then the face of an object that faces the light source but is not visible from the light source (that is, is in shadow) would still be illuminated. This is, of course, not realistic. Shadowing helps overcome this problem.
- Local illumination: shading and shadowing are called parts of the local illumination model, since even the combination of these two techniques only takes into account direct light. In the real world we observe direct and indirect light [2]. The latter means that light bounces off (or sometimes heads through) objects with some of its intensity absorbed or refracted but continue its path to illuminate other objects. With local illumination this phenomenon is not present.

- Global illumination: it is the name given to several different techniques that handle complex light reflections and refractions, with multiple light bounces. Some of these techniques only support diffuse bounces (a light bouncing off a matte surface), others specular bounces (light reflected from reflective surfaces), even others handle both in addition to caustics (an example for caustics is: a light ray goes through a glass sphere and on the other side of the sphere light patterns are formed).

- Diffuse reflection: observable in case of matte surfaces, the incoming light is reflected and scattered in many angles. This kind of reflection is viewpoint-independent. For example, in the real world it can be the light reflected by a wall painted with a matte paint.

- Specular reflection: unlike in the case of diffuse reflection, the incoming light is not scattered but usually reflected along the same angle. It can be observed on a polished steel sphere, for example. Of course, there are materials that exhibit both specular and diffuse reflections.

2.2 A brief history of computer shading, shadowing and global illumination

Realistic computer-synthesised images have been researched and written about since the early 1970s. Two of the most famous papers were by Gouraud in 1971 [3] and by Phong [4] in 1975. These papers described how to calculate local/direct illumination for single objects, without taking shadows into account. That is, according to the Gouraud and Phong shading models an object would still be illuminated even if there is another object between said object and the light source. In 1977 Blinn [5] proposed a modification to Phong shading which enhanced the quality of the specular reflections produced by normal Phong shading. This model became known as the Blinn-Phong (or Phong-Blinn) model. In 1978 Williams [6] proposed a way to generate dynamic shadows based on Z-buffer depth testing. Simple modern shadow mapping is based on similar ideas as well. Of course, since then many improved versions have been developed, for example cascaded shadow mapping, a description of which can be found in [7].

All of these models rely on local illumination, that is, only direct lights are taken into account, completely neglecting reflected lights. In order to compensate for the lack of reflected light these models usually introduce an “ambient” term, which is responsible for providing

basic illumination to parts of the scene that are not directly hit by a light source. The ambient term is fast to compute but it cannot account for the complex interreflections between different objects that happen in the real world. These reflections include, for example, complex specular effects and colour bleeding. This is where global illumination comes into the picture.

In 1980 Whitted [8] proposed a method that would allow indirect lighting to be taken into account. It is called ray tracing and it works by tracing a ray from the eye/camera to the light source and keeping track of the colour values at every bounce/reflection. Ray tracing handles specular interreflections best but it is not good at simulating diffuse reflections. Over the years many other global illumination methods have been developed, for example:

- Path tracing [9]: a technique that is similar to ray tracing, however, it is capable of producing physically accurate results. Instead of just specular reflections it is capable of modelling other phenomena (e.g. caustics) too.
- Photon mapping [10]: an improvement over ray tracing that is capable of handling extra effects compared to ray tracing, for example subsurface scattering and how light behaves when crossing particle matter (smoke, for example).
- Voxel cone tracing [11]: a relatively new global illumination algorithm, it works by creating a voxel (volume pixel – the 3D equivalent of a pixel) representation of the scene and tracing cones through it. Cone tracing is also based on ray tracing.

Of course, there are other, sometimes enhanced versions based on these (e.g. bi-directional path tracing that is based on path tracing, Metropolis Light Transport that is based on bi-directional path tracing, etc.) that can handle both specular and diffuse reflections. It can be seen that all of these techniques rely on tracing rays through the scene and then determining pixel colours from the intersections of the rays with the scene objects. However, the focus of the project and thus this report is on a classical global illumination method that works in a completely different way. It is called radiosity and the first research paper about it was published in 1984 by Goral et al. [1]. The radiosity algorithm and its history shall be explained in Chapter 3.

2.3 Motivation

One of the two main motivations for this project was to create an easy-to-use but modern radiosity renderer that relies on the GPU (Graphics Processing Unit) of a computer for most calculations in order to scale well with modern hardware. It focuses on doing one thing well: rendering scenes with radiosity. It is not an impossible task for 3D modelling software (Blender, 3ds Max) nowadays but it is not a straightforward process with them. This project proposes to fill this niche by having a simple GUI and a clear, straightforward way of getting an observable scene rendered with radiosity. It is possible to find some open-source radiosity renderers but these either use the old, fixed-pipeline OpenGL API that was deprecated with the release of OpenGL 3.0 and removed with OpenGL 3.1 for being inefficient and inflexible or use the CPU for the radiosity calculation which (as explained later in the report), leaves much of a modern computer's processing power untapped.

The other main motivation is that even though radiosity is one of the first global illumination algorithms it is nonetheless extremely underrepresented in terms of available implementation details, especially on modern systems. (This is not necessarily the case for other global illumination algorithms, see: [12][13][14], etc.) To make matters worse, many of the details that are available often contradict each other and thus it is hard and time consuming to piece together a coherent implementation approach. This raises the difficulty for people trying to learn radiosity and reduces the effectiveness of researchers who could potentially improve on the already existing radiosity techniques since it is not trivial to even get started with radiosity. This report thus aims to present a detailed description of the approaches taken and applied to each part of designing and implementing a modern, GPU-accelerated radiosity renderer. Future graphics programmers interested in radiosity could use this report as a quick-start guide, upon which they could build their own ideas. Additionally, after the conclusion of the project an in-depth technical tutorial with code samples presented wherever relevant is planned to be written and published. This tutorial would be an extended part of the implementation section of this report and would describe how the radiosity renderer built for this project is implemented from the ground-up. A sample chapter is going to be provided from this tutorial as Appendix A – "Visibility calculation with a hemicube", since public implementation notes are especially scarce about it.

Chapter 3

The theory and history of radiosity

3.1 What is radiosity?

As previously established, radiosity is a global illumination algorithm. Unlike ray tracing, it is focused on handling reflections between diffuse surfaces. To be specific, radiosity assumes that every surface is an ideal Lambertian surface, which means that every surface reflects light equally in all directions [15].

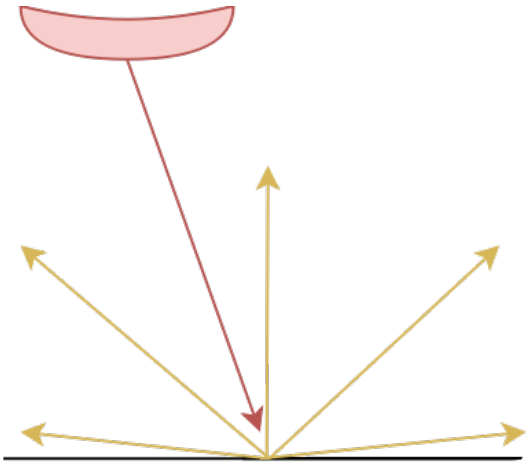


Figure 3.1: A Lambertian surface.

This means that radiosity in itself cannot handle specular reflections and is thus not good for modelling glossy surfaces. In a full-featured renderer this can be overcome by enhancing radiosity with another global illumination algorithm that can handle specular reflections, for example, ray tracing. The advantages of radiosity are, however:

- It is viewpoint independent: once lighting for a scene has been calculated, it can be stored in memory and displayed continuously. The observer can freely look around the scene without light values having to be recalculated. This makes radiosity incredibly advantageous for pre-calculating lighting for static scenes and presenting this scene to the user. This is not true for every global illumination algorithm. Path tracing, for example, has to completely restart calculations whenever the viewpoint changes.

- Colour bleeding is observable: colour bleeding happens when a direct or strong indirect light first hits a coloured surface and then another surface – for the sake of an example: let the second surface be white. When reflecting, the coloured surface absorbs a spectrum of the light, the parts that our eyes would not see in the end: if a white light hits a red surface, the surface absorbs every component of the light, aside from red. It can be seen that the reflected light is going to be what we would call red. If this red light hits a white surface, it is going to make it appear slightly red. Radiosity handles this complex interreflection.

- Soft shadows do not require extra computation: in real life shadows are rarely hard shadows – that is, if an object casts a shadow it is usually not completely dark, but some reflected diffuse light makes it lighter. This means that the colour of the object "behind" the shadow can still be seen in addition to the shadow itself. Since radiosity handles diffuse light bounces, it replicates the same effect.

Unfortunately, radiosity has its fair share of drawbacks too. These, along with a more detailed history of radiosity shall be touched on in the next section.

3.2 The history and description of different radiosity techniques

The high-level way of how most radiosity techniques work is that the surfaces in a scene are subdivided into smaller "patches". These patches accumulate and store the amount of light arriving at them, which is going to be distributed ("shot") back into the scene again. Usually the energy stored in a patch is uniform: the energy does not vary over the surface of one patch. It is necessary to calculate how much a patch sees another patch, since this also dictates how much energy is transferred between the two patches. This is called the form factor between two patches:

$$dF_{dA_x \to dA_y} = \frac{\cos\theta_x \cos\theta_y}{\pi r_{xy}^2} V(x,y) dA_y$$

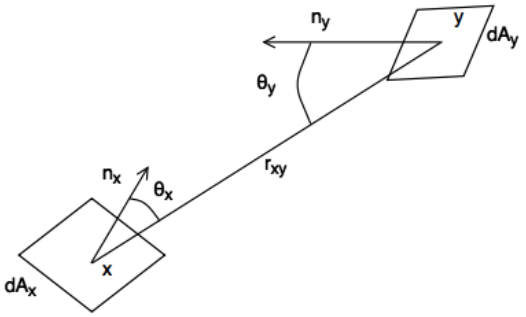


Figure 3.2: The form factor between two differential elements (patches). Both the formula and the image are based on [16]. r_{xy} is the vector from one patch to the other, θ_x and θ_y are the angles between r_{xy} and the respective normal vector of the patches. dA_x and dA_y are the areas of the respective patches. V(x,y) is the visibility term.

Let us take a detailed look at how two "pure" radiosity techniques solve these issues, followed by a brief look at two newer, hybrid radiosity methods.

1984, Goral et al.: Classical Radiosity [1]: The original radiosity technique proposed solving the diffuse part of the rendering equation (for more information about radiosity and the rendering equation see [17]) by solving an NxN matrix (where N = number of patches) of linear equations composed of the form factors and distributed energy of each patch. In order for this solution to work the surfaces in the scene had to be subdivided into smaller surfaces (the previously mentioned patches). Then form factors had to be determined for each patch. This was done by integrating over a hemisphere placed over the patch in question. This hemisphere

represented everything a patch could see in the scene. Finally, the aforementioned NxN matrix had to be solved. In practical terms, this means that a patch distributed its energy to all of the other patches visible from the given patch. This had to be done for every patch (which means solving every line in the NxN matrix) and the final radiosity value was returned for every patch in the scene, which could be displayed on the screen.

This method allowed scenes to be rendered with complex inter-object diffuse reflections. For example, colour bleeding was observable and soft shadows were "free": as previously stated a well-implemented radiosity renderer would be able to place soft shadows correctly without any additional calculations. It is also view-independent. For a static scene the matrix only had to be solved once, then the observer could freely fly around the scene without any further complex computations. This is not true for ray tracing or path tracing, for example.

The downsides of classical radiosity were: incredibly high initial computation costs – it is needed to produce, store and solve an NxN matrix so it scales quadratically with the number of patches. In a modern medium-complex scene even if we take one triangle as one patch it would mean computing a matrix that has tens of millions or even significantly more elements than that. For example: the Crytek Sponza scene [18] has 262267 triangles, which makes it a fairly complicated but not overly complicated scene (e.g. the Unigine Superposition benchmark [19] released in April 2017 has multiple millions of triangles on screen at once). For Crytek Sponza a classical radiosity solver would have to produce and solve a 262267 x 262267 matrix which is just below 70 billion elements. Since it is a static scene it would not be impossible to do, but it would be slow and highly impractical. Another issue ties in with this, however, which is using this method for dynamic scenes. Even though some information from the previous rendering could probably be retained, some parts of the matrix would have to be recomputed nonetheless. This is unlikely to be possible with a good framerate.

Classical radiosity is capable of generating realistic diffuse reflections but it is computationally expensive and it also does not scale too well for modern renderers. Fortunately, in 1988 a new approach was published which allowed scenes to be updated before the end of the whole radiosity computation.

1988, Cohen et al.: Progressive Refinement Radiosity [20]: The point of this new radiosity approach was to simplify the calculations that are present in classical radiosity. This was achieved by transforming the algorithm so that the form factors are computed "on-the-fly".

The form factor calculation was also changed. Instead of the hemisphere method a hemicube is placed on the patch and form factors are calculated this way. This approach was described by Cohen et al. in 1985 [21]. Two other improvements were also made: instead of the "gathering" approach used by Goral et al. [1] which meant that a patch first received all the light from the visible patches Cohen et al. use a "shooting" method – one patch distributes all its energy to the visible patches ("receivers") and thus updates the radiosity values of multiple patches. If the shooter with the highest energy is chosen then the solution converges quickly, since these patches contribute the most to the lighting of the scene. The second improvement is the (re)introduction of the ambient term. In this case the ambient term would try to give a rough approximation of the end result during the early stages of the algorithm and would fade out as the algorithm progresses, so instead of a dramatic change (from a completely dark scene to a lit one) it would start with a naïve (ambient) lighting solution then would gradually switch to the radiosity-computed solution.

This radiosity approach is suitable for implementation in a modern renderer and also seems to be the most appropriate one for the current project, thus details about its implementation will be given in a later section. For the sake of completeness two other, newer techniques are briefly described, even if they do not have a lot in common with the first 2 radiosity algorithms.

1997, Keller: Instant Radiosity [22]: this algorithm approximates diffuse and specular radiance by creating virtual point lights (VPLs) where a light ray hits an object. This VPL is going to emit light as if it was reflected from a given surface. For example, if a ray from a white light hits a green surface a green VPL is created where the light ray hit the surface. It is rather dissimilar to the first two radiosity algorithms but it does support colour bleeding and specular reflections too. It is sort of like radiosity and ray tracing mixed together with some new ideas.

2014, Mara et al.: Deep G-Buffer Based Approximate Radiosity [23]: This radiosity method is implemented with the help of the modern programmable geometry shaders on the GPU. Mara et al. also describe how to integrate this radiosity method with other global illumination methods (ray tracing) and ambient occlusion.

Let us get back to the task at hand and take a look at how progressive refinement radiosity could be implemented.

The CPU method: The main advantage of this method is that familiar CPU programming paradigms and data structures can be used. In this case the traditional geometry-based patch creation would be used (use one triangle per patch initially, even though those could be subdivided if more detail is needed). Unshot energy would be stored per triangle and if a list is kept of all triangles this energy could be distributed to all other visible triangles by looping through the list and calculating their form factors. This could be done in two ways: Manually, using a surface normal computed from the triangle's vertices and implementing a simpler version of a ray tracer along the lines of [24] or offloading this calculation to the GPU. The former would possibly require lots of complicated calculations as well as the implementation of a ray tracer and the latter would have to deal with the CPU-GPU communication problems that are going to be mentioned in the GPU section of the report.

In this case the irradiance values (the values that the original RGB colour of the triangle are going to be multiplied with when rendering) would be stored as a simple RGB three-component floating point vector and used to modify the colour values after interpolation.

The CPU-based technique would work in cases where the material colour is directly stored within vertices but would struggle with cases where we apply textures, since getting texture data on the CPU and adding that to individual vertex colours (which is something the above described method relies on) is not a trivial task. It would, however, work with scenes that have no UV coordinates specified per vertex.

It is apparent that the CPU-based technique is rather inelegant and since it mostly ignores the additional computational power at disposal provided by the GPU it is also rather archaic. It has its pitfalls and limitations but due to its being closer to CPU programming it would likely to be easier to implement. However, this CPU-based implementation plan is not favoured since doing it this way would mean giving up on the benefits of using the GPU. This would be a huge waste of the resources available on a modern computer.

Additionally, GPUs are rapidly evolving nowadays. Nvidia released the flagship consumer card of their new Volta architecture, the GTX Titan V on 7 Dec 2017. According to PCGamer [25] this card is capable of handling nearly 15000 GigaFLOPS (FLOPS = floating point operations per second). The two previous flagship cards, the GTX Titan Xp and the GTX 1080 Ti (released in April 2017 – the PCGamer site mistakenly gave the older GTX Titan X release date in [25], see [26] for the correct one – and February-March 2017, respectively) were both

capable of around 11300 GigaFLOPS. This is an increase of about 32% in the space of less than a year. CPUs are evolving at a great pace too (for example, see [27]) but confining a solution to just one part of the hardware at our disposal would be sub-optimal.

The final reason against the CPU-based approach is that the different steps of radiosity calculation can be excellently ported to the GPU, taking advantage of their massively parallel way of working. In order to make this apparent a whole chapter is going to focus on how GPUs work and how they are different compared to CPUs. In order to provide information about GPUs in one place and prepare the reader for the radiosity implementation section, the aforementioned chapter is also going to describe how GPUs are programmed and some of the more advanced techniques a programmer can perform on them.

Chapter 4

Design & specification

4.1 Graphics application design

Designing a GPU-accelerated graphics application is rather different compared to conventional desktop or mobile applications. In those cases, the backend and frontend can be separated and individually designed, later programmed. In the case of pure graphics applications (so not counting games and game logic) separating the two are not necessarily straightforward. The backend is usually responsible for calculations and business logic. This information is then passed to the frontend, which can display that to the user. The renderer part of a graphics application has to handle both duties, since most calculations and functions ultimately serve one purpose: to output correctly coloured pixels at the correct locations on the screen. Consequently, the result is a "transparent backend" that includes business logic but is ultimately responsible for visuals. This backend also contains pieces of code running on the GPU, called "shaders". This is going to be examined in more detail later on but let us take a look at the requirements of the program now.

4.2 Requirements

The main goal of this project is to allow the user to load an arbitrary scene (discounting one hard limitation and a few recommendations, described later), observe the virtual scene before the global illumination calculation, place point lights around the scene, start the radiosity

calculation and also observe the scene after the radiosity updates are finished. The requirements are described in a way a normal execution of the program would happen.

First, the user shall be greeted by a simplistic Welcome/Start screen. An elaborate frontend is not part of this project and thus UI screens should be kept as simple as possible. This screen shall allow users to: start the renderer; select the object file to open by opening a file selector window on the click of a button; display the file path of the selected file; open a settings menu where some quality/performance trade-off settings can be configured and if the user wants to, close the program. Once the user launches the renderer part itself, the following functionalities should be available: the user can navigate the scene from the perspective of a virtual camera. This, of course, means controlling where the camera is within the scene with the keyboard and the mouse. The user can switch on a small amount of ambient light in order to be able to explore the scene before the radiosity calculation. The user can place point lights[1] around the scene. After the desired light sources are placed the user can manually initiate the radiosity process and depending on the settings, stop it at will or when the shooting mesh is finished shooting. The user can observe the scene and see what it looks like after each iteration of the algorithm.

These are the user-facing features. In addition to these, the application should also have a frame time counter and a way to reload shader programs at runtime. The former is useful for performance profiling and the latter is useful for runtime debugging.

As a non-functional requirement, most of the radiosity calculation should be done on the GPU. This is to ensure that the resources of a given system are utilised as much as possible and said resources are used for what they do best, thus increasing the performance of the software. It is also good for the ability to automatically scale on newer hardware. Since the project has no access to any modern and/or high-quality hardware no exact performance requirements are set. Radiosity is an expensive algorithm and, while it can be optimised, it is still unlikely to run well on old hardware. Another common non-functional requirement is security, since systems (at banks, insurance companies, etc.) often handle personal data. This project, naturally, does not deal with any personal data. It, however, deals with the computational power and the memory of the host computer. Through OpenGL it is possible to create massive-scale memory

[1] A small correction: for the sake of simplicity, throughout the report the light sources used by the program are referred to as "point" lights. This is not entirely accurate. Point lights have an infinitely small area, but the light sources in the renderer have an area just like every other object in the scene. They are, thus, area lights that try to simulate point lights.

leaks that – while unlikely to cause any lasting damage – can still cause the host computer to slow down to a crawl, which should be avoided.

These requirements expressed in a more formal way:

1. Frontend requirements:
   (a) The user must be greeted with an initial window on start.
   (b) The user must be able to start the renderer.
   (c) The user must be able to select a Wavefront .obj file of the desired scene (with a few limitations).
   (d) The frontend must display the filepath of the selected file.
   (e) The user must be able to open a ”Settings” menu/window.
   (f) The user must be able to adjust a few settings (renderer resolution, radiosity lightmap resolution, attenuation type, how the scene is updated and the use of lightmap filtering and multisampling) by either selecting a value from a drop-down box or by ticking a box, depending on which is the more appropriate way for the option in question.
   (g) The user must be able to close the program.
2. Renderer requirements:
   (a) The renderer must load the scene file provided by the user if the scene is well-formed (i.e. adheres to the limitations of the renderer described later).
   (b) The user must be able to navigate the loaded scene with a virtual camera.
   (c) The user must be able to switch on some ambient light.
   (d) The user must be able to place point light sources before the radiosity rendering is initiated.
   (e) The user must be able to initiate the radiosity process.
   (f) The user must be able to observe the scene rendered with radiosity.
   (g) Depending on the settings, the user might be able to stop the radiosity calculation at will.

(h) The renderer must be able to render well-formed scenes with radiosity and the radiosity-rendered scene must exhibit the typical features of radiosity (soft shadows, colour bleeding, view independence) where appropriate.

3. Developer-oriented requirements:

   (a) The application must have a frame time counter.

   (b) The programmer must be able to reload shader programs at runtime.

4. Non-functional requirements:

   (a) Most of the radiosity calculations have to run on the GPU.

   (b) The application must avoid critical, large-scale memory leaks.

4.3 APIs and Libraries

Libraries are pivotal in most bigger-scaled software projects and this project is no exception either. Before we take an in-depth look at the design itself, let us examine the tech stack.

As mentioned previously, the main part of the project is OpenGL 4.5. This is not a programming language in itself, just an API, so it needs a programming language it can build on. The most convenient choice is to use C++, since OpenGL directly provides functions for it. Other languages need bindings in order to be compatible with OpenGL. The newest release of C++ was chosen for the project, C++ 17. It should be the most polished and feature-complete version of the language so there is no reason not to use it.

While OpenGL is not a language on its own, it does support its own shading language for code that runs on the GPU. It is called GLSL. It is similar to a more graphics-oriented C++. In order to be able to seamlessly work with C++ OpenGL needs a few libraries. The ones used, in no particular order, are:

- GLFW: This library handles window and context creation for OpenGL.
- GLEW: OpenGL Extension Wrangler, it provides easy access to OpenGL extensions.
- GLM: A mathematics library that aims to port most of the features from GLSL to C++, it allows code to be written in C++ that can easily interoperate with GLSL.

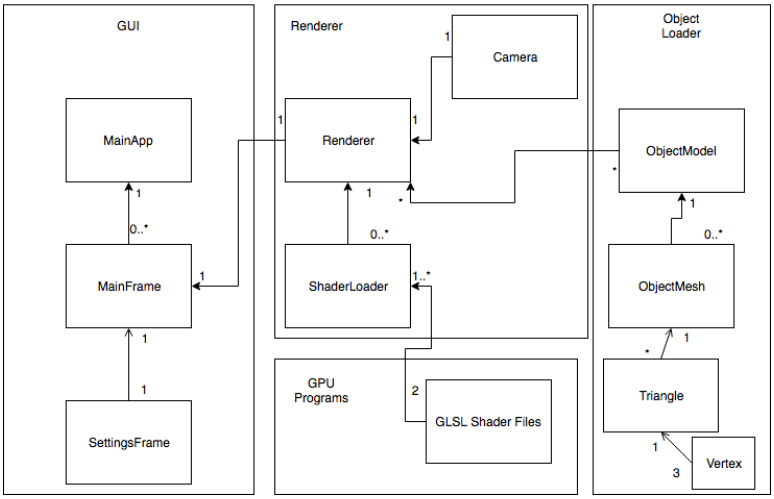
Two other libraries are also in use, to deal with image and object loading:

stb_image.h is a simple header-only library that handles image loading. It is a simple, easy-to-use image loader. The object loader is Open Asset Import Library (Assimp). It is a heavyweight object loader that can handle a myriad of object file formats and also do post-processing on them. Its main usage in the program is to simply load the vertex data contained within the object file, along with the associated textures/colours. The frontend also has its own library, called wxWidgets.

4.4 Design

The project can be split into 3 main parts: the frontend, the renderer and the shader files. The frontend is only a few files, roughly corresponding to the individual windows the frontend part of the program has. The renderer contains the parts of the code that run on the CPU. These can be split into 3 smaller chunks. The Camera and Renderer classes belong to the main renderer. Their purpose is to directly handle what is being drawn on the screen, including the radiosity process and most of the functions supporting the radiosity process. The Triangle, ObjectMesh and ObjectModel classes and the Vertex struct belong to the Object Loader. These are the ever-increasing building blocks of a virtual scene. 3 vertices make a triangle. Triangles make up bigger objects in OpenGL, such as meshes. A mesh, for example, can be a chair. Multiple meshes make up a model, which is the whole loaded scene in the renderer. It can be a fully furnished room with walls. The last part of the renderer deals with the GPU. OpenGL does not provide a default way to load GLSL shader files. The ShaderLoader class handles this functionality. It also handles the compilation of these shaders into shader programs and provides a more convenient way to set uniform variables. Finally, there are the shader files, both vertex and fragment shaders. These are usually short files with a few functions each. They are going to be detailed whenever they are mentioned in the implementation section. Here is a simple UML diagram to illustrate the described system:

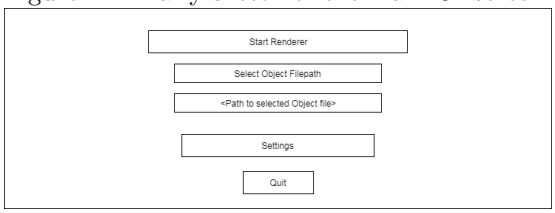
Figure 4.1: A simple UML diagram of the architecture of the system.



4.5 Frontend

As previously stated, an elaborate user interface is not part of the project. It should be a quick gateway to the renderer allowing the user to set a few parameters (for example, the scene object model to be loaded). Since the project is written in C++, it is possible to interface with the window creation utilities of the operating system through it but that would definitively tie the project to a single OS. Explicit multiplatform compatibility is not an aim of the project but it should only use OS-specific tools if there is a direct benefit to them. As already mentioned, the project uses a library that is cross-platform compatible, called wxWidgets, for the UI. The advantage of this library is that it closely mimics the look of the windows of the host OS and thus programs written with this library should not stand out from native applications/windows. Here is a quick sketch of the expected UI:

Figure 4.2: Early sketch of the main UI screen.



The GUI is not revisited again in the report, so let us take a look at the implemented UI:

Figure 4.3: The implemented main UI.

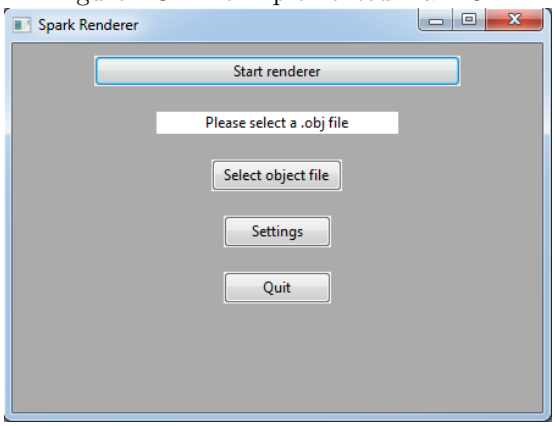
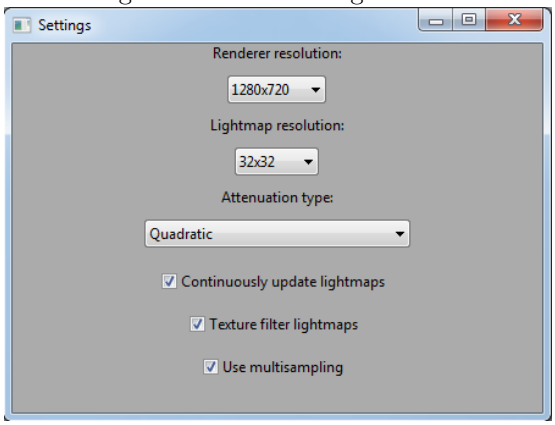


Figure 4.4: The settings screen.



From this point the report mostly concerns itself with GPUs and graphics programming, so the next chapter gives a general overview of both.

Chapter 5

GPUs and GPU programming – a primer

5.1 A short introduction to GPUs

As already mentioned in the report the name "GPU" simply stands for Graphics Processing Unit. As the name implies these units are mostly used for drawing 2D or 3D images to the screen. Unsurprisingly, they are optimised for this task, however, they can be also used for General-Purpose GPU (GPGPU) computing. Discrete (non-integrated) GPUs have their own RAM but in some cases they can also access the much slower RAM of the CPU.

Modern GPUs generally make use of driver programs that are installed on their host machine. Among other things this driver is responsible for taking commands and other information from the CPU for the GPU; coordinating the CPU and GPU; handling data readbacks between the two units and naturally, it is also responsible for drawing images on the screen which are made from millions of pixels on modern display devices.

In order to be able to program GPUs, there needs to be a way to communicate with the driver itself. This can be done through graphics Application Programming Interfaces (APIs). Such APIs include Direct3D/DirectX, Vulkan or the one used in this project, OpenGL. The advantage of OpenGL is that it is a cross-platform API (as opposed to Direct3D which is Windows exclusive) and even modern versions of it run on a broad range of hardware. In case of Nvidia GPUs even the 400 series cards provide support with up-to-date drivers up to the

latest OpenGL version, OpenGL 4.6. The 400-series was released in 2010. Vulkan is a new graphics API and it does not support legacy hardware, including the test GPU for this project, so Vulkan was not an option. It means that the decision for the API was between DirectX or OpenGL. Even though the program built for this project does not explicitly target cross-platform compatibility, OpenGL makes it easier to port the program in the future so version 4.5 of it was chosen. Next, let us examine how it works.

5.2 OpenGL

OpenGL was originally released in 1992. OpenGL versions below 3.0 used a so called "immediate mode" with a fixed pipeline. This mode highly restricted the programmability of the GPU and also had performance issues. One of the upsides of this mode was the fact that it was easier to program, since OpenGL itself handled most of the heavy lifting. This mode was deprecated in version 3.0 and removed in 3.1. OpenGL versions starting from 3.3 are usually considered "modern" and new programs are recommended to use 3.3 or newer versions. These versions are split into "core" and "compatibility" profiles, where the compatibility profile includes the previously removed fixed pipeline functionality. Needless to say, the core profile is recommended for new applications, since this uses the programmable pipeline. It exposes stages of the rendering pipeline to the programmer. The term "pipeline" hides a complex, multi-stage system that transforms and processes data that was sent to the GPU and generally produces a set of pixels (which are also called fragments).

In the next section the major parts of this pipeline are examined. Afterwards the way of programming this pipeline is explored.

5.3 The OpenGL pipeline

This section is a short overview of the functionalities relevant to the project. More information can be found at [28].

In order to start the pipeline, OpenGL expects a set of vertices that define points in 3D space. It has to be mentioned that OpenGL is capable of linking these vertices in multiple ways in order to create "primitives", of which are a few: no linking i.e. display the vertices

separately, create lines out of the vertices and the mode that is used within this project: create triangles. OpenGL simply creates triangles based on the order of the vertex data sent to it: every 3 vertices in a row create a triangle.

The first stage of the pipeline is the vertex shader. This is a programmable stage that can modify every single vertex sent to it and all their supplied attributes, one by one. When this stage is done, every input vertex is sent forward in the pipeline as an output vertex.

Afterwards there are three optional programmable stages: the tessellation control shader, the tessellation evaluation shader and the geometry shader. The first two are capable of generating new vertices and thus making the geometry in a scene more complex and detailed. The geometry shader allows a programmer to modify primitives instead of individual vertices.

After these stages comes a vertex postprocessing stage, which is responsible for clipping, the viewport transformation and the perspective divide. The perspective divide is going to be relevant in the later sections and shall be further elaborated on later.

The next two (non-programmable) stages are primitive assembly and rasterisation. The former is responsible for creating a sequence of primitives (e.g. triangles) out of the supplied vertices. It has to be highlighted that the geometry shader coming before this stage already operates on primitives. In order to accommodate this feature, if the tessellation or geometry shaders are active then a limited, early primitive assembly stage is run before the vertex postprocessing stage. The primitive assembly also includes a face culling stage. The point of this is to determine if a primitive is facing towards or away from the viewpoint. In case of opaque/solid objects, these can safely be not rendered since the user normally cannot see inside such materials, saving processing power while not affecting the scene that is visible to the user. Face culling simply works by checking if the vertices of a primitive run clockwise or counter-clockwise. By default, faces with clockwise vertices are culled/discarded.

Rasterisation takes the remaining primitives from the previous stage and turns them into fragments (which are roughly equivalent of pixels). These fragments are then passed onto the last programmable part of the pipeline.

The fragment shader is an optional stage in the rendering pipeline but most commonly it is not skipped. It is used for calculating and assigning colours to individual fragments. A major feature of the fragment shader is that interpolated data is available for every fragment:

every vertex holds some data when it enters the pipeline. The primitive assembly creates a series of triangles. These triangles have 3 vertices, each with one or more fields of data (called "attributes"). These different data points are interpolated over the surface of the triangle between the 3 vertices. The rasteriser creates fragments out of these triangles but it makes the interpolated data (from where the fragment was generated) available in the fragment shader, which the programmer can access and use for calculations. This is also the stage of the pipeline where textures can be read. Another very important feature is the ability to set the depth value of a fragment. The fragment shader in the normal case can output a colour value, which is going to be the colour of the fragment that is to be displayed on the screen, if it is not discarded by one of the per-sample operations that conclude the pipeline.

The only per-sample operation used in this project is the depth test. Through an approach called the Z-buffer (to be explained in the next section) it is tested if there is a fragment closer to the user than the current fragment. If there is, there is an object between this fragment and the user. If only opaque surfaces are considered then it is known this fragment would not be visible, thus it can be discarded.

After this step the processed data is output to the render target and the pipeline concludes. The way to control this pipeline has been mentioned only superficially so far so let us take a deeper look into that.

5.4 Programmability

The previous section already implies that GPUs are fundamentally different from CPUs and thus the way they have to be programmed is different as well. CPUs generally execute commands in a sequential manner but allow the programmer to explicitly create concurrent sections within a program (usually called threads). GPUs, however, are stream processors. For our current purposes this means that they are capable of a limited form of SIMD (Single Instruction – Multiple Data) parallel processing on a massive scale [29]. The upside of this is that they are incredibly efficient for computing so called "embarrassingly parallel" problems. If we decompose embarrassingly parallel problems into smaller sub-problems then these sub-problems or their results do not depend on each other at all. This means that the sub-problems can be freely done in parallel [30]. If a look is taken at a computer-rendered image it can be seen that it is made up of possibly millions of small elements, called pixels. Fortunately, these pixels do

not depend on each other and the GPU pipeline can parallelise most of the steps leading to the production of these individual pixels.

Instead of using conventional object-oriented principles OpenGL operates as a complex state machine where objects can be created, configured, bound and destroyed by API calls. In order to start the rendering pipeline a set of vertices have to be supplied to the API. This can be done by Vertex Array Objects (VAOs) and Vertex Buffer Objects (VBOs). Through these objects can the vertex data be specified and the way how OpenGL should handle it be configured. This brings us to the first programmable stage of the pipeline, the vertex shader. As an early note, the project only uses vertex and fragment shaders so the other programmable shaders (the two tessellation shaders and the geometry shader) are not going to be described in-depth in this report.

The vertex shader is a small program (also known as a "kernel" program) that runs on the GPU and as its name implies, operates on individual vertices. This shader can take as its input every attribute a vertex might have – 3D (usually model-space) coordinates, vertex normal vectors, UV coordinates, etc. It also has access to "uniform" variables. These variables are declared before the execution of the shader program and do not change between executions. These can be as simple as an integer but commonly vectors and matrices are also used. The vertex shader can have multiple outputs that get passed to the next programmable shader stage, which, in our case is the fragment shader. The vertex shader, however, has one required output variable: gl_Position. This is a four-component floating point vector that determines where the vertex should be rendered. Do note that while normally rendering a scene a vertex might not appear on the screen at all, for example if it lies outside the field of view of the viewer or if it is obstructed by an object that is closer to the camera.

After the vertex shader stage comes the fragment shader. It operates on individual fragments/pixels and this stage is where textures are commonly used. Because of that, it would make sense to look at what textures are.

Since textures are heavily used during the project in multiple (and sometimes seemingly unorthodox) ways it is important to realise that textures in OpenGL are not more than arrays of numeric values. These numeric values can be used to represent colour or lighting information and are referred to as "texels". Renderers generally use them to provide details to surfaces in

the scene without having to overcomplicate the geometry of the scene. For that purpose many different texture types exist, of which are a few:

- Diffuse texture: represents the diffuse colour of a surface.
- Specular texture: represents the "shiny" parts of a surface. With these, parts of a surface can be made to have specular reflections while other parts not.
- Normal map: used to simulate small bumps on surfaces.

And many other types. Since radiosity concerns itself with diffuse surfaces, diffuse textures are used in this project.

Textures are commonly rectangular but objects within a scene can be nearly any shape. That raises the question: how are textures applied (mapped) to a surface? The answer is UV mapping.

In OpenGL textures are handled within a 2-dimensional coordinate system that ranges from (0, 0) in the bottom left corner to (1, 1) in the upper right corner. UV coordinates assigned to a vertex simply represent a point on a texture where this vertex lies. When OpenGL constructs triangles out of individual vertices the UV coordinates between the vertices are interpolated over the triangle. The end result is that a certain part of the texture is going to map onto the triangle. Generating UV coordinates is usually the responsibility of a modelling software (such as Blender).

Before we return to the fragment shader, let us examine another functionality that is relevant at this stage: depth testing.

A 3D renderer operates in the 3D space but computer monitors are only 2 dimensional, hence from a given viewpoint there might be objects that are obstructed by other objects. The Z-buffer (or depth-buffer) algorithm is used for determining visible objects. During the vertex shader stage every vertex gets a depth (or Z) value. The pipeline in the end produces depth values for every pixel, ranging from 0 (nearest) to 1 (farthest). When a pixel is first written at a certain position its corresponding value in the Z-buffer is updated with the depth value of this pixel. This is, by default, not terribly interesting. However, if the rasteriser produces another pixel that maps to the same location as the previous pixel then the value mapping to this position in the Z-buffer is checked. If the depth value of the current pixel is smaller than

the retrieved value (so it is closer than the previous pixel) it is allowed to overwrite the previous pixel value. Otherwise, however, the pixel is obstructed and can be safely discarded.

That leads back to the fragment shader. In essence, computer graphics boils down to putting the correctly coloured pixel at the correct place on the screen. The fragment shader is the stage that allows the programmer to ultimately do that. The fragment shader has access to the vertex shader's interpolated output data; it can do texture lookups to fetch the colour value of the texture through the interpolated UV coordinates; it can set the depth of the fragment to something different compared to the output of the vertex shader and most importantly, it can output a 4-component float vector that represents Red, Green and Blue (RGB) colours of a fragment and an alpha channel that is used to measure opacity. Radiosity generally deals with fully opaque objects so our alpha is fixed to 1.0 (fully opaque).

Since interpolated data is available for the fragment shader including, but not limited to: interpolated normals, interpolated world-space position, etc. it makes the fragment shader ideal for illumination calculations. In the case of simple local illumination models (Phong shading [4]) all the illumination data can be calculated in a short fragment program. As we go towards more complicated models (shadow mapping) and global illumination (radiosity) the number of required fragment programs and their complexity ramp up exponentially.

Some advanced but not always well documented GPU programming solutions are also used in the project. These shall be described shortly. However, one crucial feature of 3D renderers has so far been omitted. Namely, coordinate systems.

5.5 Model/World/Camera/Clip spaces

In this section four coordinate systems are described, since a basic renderer makes use of these four. A few others are used in the project but their descriptions are going to be left to when they are first used.

The most basic coordinate system used in the renderer is the coordinate system of a model/mesh. This is usually called model-space. It describes the size and shape of a mesh, for example that of a chair and is usually used in modelling software, when assets are created for a program.

The second coordinate system is the world-space coordinates. This describes where the vertex coordinates are located in relation to the whole scene. It is usually obtained by translating, scaling and maybe even rotating the vertices defined in model-space coordinates. This is called the world-space transformation and can be obtained by multiplying the 3D model-space coordinates of vertices with a matrix constructed from the translation/scaling/rotation operations. This matrix is called the "model" matrix. Using this matrix can be thought of as placing the aforementioned chair inside a room.

The third coordinate system (camera-space or eye-space coordinates) is important when the concept of a virtual eye/camera is introduced. It has to be noted that, unexpectedly, the camera itself does not move within a renderer, at least not in a conventional sense. Instead everything else in the scene is subjected to transformations. This effect can be achieved with a "view" matrix. In order to be able to construct this view matrix it has to be known where the camera is within the world and the direction vector it looks along. The orientation of the camera also has to be decided. It can be done by specifying an "up" vector. As long as a system does not implement a "camera roll" function this up vector can point towards the positive Y vector, without any negative side effects. The construction of this matrix can be done manually or outsourced to a commonly used OpenGL library called GLM. It stands for OpenGL Mathematics. If world-space coordinates are multiplied with the view matrix they are transformed into camera-space coordinates. This is a representation of the scene as it is seen from the camera. However, this is not enough to accurately represent a scene since we do not account for perspective. In the real world, objects that are farther away appear smaller. In order to represent this the help of a projection matrix is needed.

There are two major kinds of projection matrices. Orthographic and perspective. The former is a simpler kind of projection where foreshortening is not observable. It is useful for use cases where exact distances and lengths are needed, for example in modelling applications or engineering schematics. However, this is never explicitly used in the project. Perspective projection is what is sought, since this exhibits the natural foreshortening effect. Aside from the foreshortening effect, the perspective projection matrix also handles the field of view (which gives how many degrees the user can see from a viewpoint), the aspect ratio of the image and the near and far clipping planes. If pixels get rendered too close to the viewpoint they can produce artifacts, back projection and possibly divisions by 0. Avoiding all of these is desirable. This is where the near clipping plane comes into the picture: the matrix cuts off every pixel

that is rendered at or behind this plane. The logic behind the far clipping plane is similar. Some parts of the scene are rendered so far from the viewpoint that they could not be rendered properly, thus the far clipping plane cuts off everything that is farther than that plane.

After having done every step the end result is a view frustum. This contains every part of the scene that can be seen from the camera. One thing to keep in mind is that matrices have to be multiplied in reverse order for the desired effect. In more concrete terms, in pseudocode it would look like this:

$$projection * view * model * vertexPos$$

5.6 Debugging

Due to the parallel, stream processor nature of GPUs debugging is a rather difficult task while mostly not being able to rely on any of the conventional CPU debugging techniques (which are, for example: printing variable values to the screen, using a debugger to trace the execution of the program, etc.). Naturally, these can be used on the CPU side of graphics applications but bugs can be caused by the CPU, the GPU and occasionally both at the same time. To further complicate matters, it is mostly left to the programmer to keep track of the state of the OpenGL state machine. The programmer also has to actively check for OpenGL errors by defining a callback function for it and even then, OpenGL silently ignores some abnormal commands. This can be seen in the case of glDeleteTextures(), for example. If it is called with an argument that does not correspond to a texture then this function silently does nothing. It shows that the best OpenGL "debugging" strategy is to code carefully and only after having obtained a good understanding of what is going on behind the scenes. Of course, even the best programmers make mistakes occasionally and there are a few techniques that can help.

Technique 1 - Staggered execution: This one is useful for radiosity since it relies on one algorithm being executed a large number of times. The execution of the radiosity algorithm would halt after 1 iteration (or another specified number). In this state the scene could be observed and evaluated.

Technique 2 - Print framebuffer to a part of the screen [31]: Since computation results in framebuffers are relatively hard to trace, it is useful to have a way that allows the end result within the framebuffer to be observed. The scene can be drawn normally first, followed

by a screen-aligned quadrilateral that only maps to a part of the screen, for example the upper right corner. This way it is possible to have a continuous render of the scene and a continuous render of the framebuffer to be observed.

Technique 3 - Display fragment data for the whole scene: Instead of outputting a final colour value from a fragment shader it is possible to output another variable instead. For example, in order to check if the geometry of the scene is correct it is possible to output the normal vectors of the fragments before the final render. This would visualise every fragment normal within the scene. Two things have to be kept in mind, however: the bit precision of the default framebuffer can cause issues if the values only change very gradually, since the output colour would be hard to tell apart. The second is, since technically colour values are output, they cannot be negative, while normal vectors, for example, can be. It is rather trivial to map normal values to the [0-1] float range colour values are usually expected to be in, but this step should not be forgotten.

Technique 4 - External graphics debugging programs: There are many, sometimes GPU-vendor specific programs (Nvidia Nsight, AMD CodeXL, etc.) but the one used in this project was a standalone, open-source debugger called RenderDoc. It works by capturing and launching the executable of an OpenGL/Direct3D11/Direct3D12/Vulkan application and launching it. The programmer can command RenderDoc to capture individual frames of the application. These frames can then be inspected individually. The available information they contain is usually a part of the OpenGL state. This can be, for example, the state of the pipeline, the input and output of the vertex shader, the meshes drawn, the calls to the OpenGL API and the feature most used in this project, the ability to view active textures. Radiosity heavily depends on texture-space rendering and the output of multiple framebuffers. These can be traced in the rendering application itself as well, but not in any trivial way. RenderDoc provides a "Texture viewer" functionality that lets the programmer easily access all of the textures in use during the captured frame. This makes it easier to spot mistakes (such as a framebuffer not being written to at all due to a programming fault) and evaluate visual artifacts.

Technique 5 - Problem solving through experience: This one is a rather obvious "technique" but it can be hard to qualify, describe and search for fixes for visual artifacts, especially for someone not particularly experienced with graphics programming. The fact that graphics APIs evolve and change massively over their lifetimes magnifies this problem. Additionally, it is not always easy to find enough material about a particular sub-domain, especially

for a given version of a graphics API. In conventional Java or C++ programming, generally solutions are forward-compatible. That is, an old answer to a given problem can be used for newer versions of those languages but this is not always true for graphics. Sometimes the solution can only be reached through personal experience and the range of possible graphical faults the programmer has seen.

5.7 Advanced techniques

Technique 1 - Framebuffers: By default, the end result of an OpenGL draw call goes to the screen of the user. This is also called the default framebuffer. The programmer can declare other framebuffers for so called "off-screen" computations. In more complicated systems these are computations that the user does not need to directly see. They often form sub-parts of complex graphical effects. These framebuffers can hold multiple colour textures and a depth texture (or its write-only, faster equivalent, a renderbuffer). These framebuffers work similarly compared to rendering to the screen, aside from a few differences. The colour textures in the framebuffer have a programmer-defined precision, thus they are capable of storing floating point data that the default framebuffer would not be able to give back faithfully. They can also store negative values. Aside from outputting values into one texture, a framebuffer is capable of handling multiple render targets. This is more significant as a performance improvement, since if results that have identical layouts but different data are intended to be output then it can be done within one pass. It requires a specially written fragment shader that outputs data into different textures instead of doing this through multiple passes and multiple fragment shaders. At this point it is also worth noting that textures can be either write-only or read-only. Unfortunately, it is not possible to have "read/write" textures. It is possible to get around this limitation by using separate textures but it is not natively supported by OpenGL. Framebuffers end up containing textures with the rendered data after the draw calls are finished. The CPU can also read back this data that is stored on the GPU, which is going to be used a few times during the project. This operation is slow and should only be used when necessary.

Technique 2 - Shadow maps: In order to create shadows a technique called "shadow mapping" is commonly used. This technique introduces another coordinate system to the four already examined ones. This coordinate system is a modified camera-space, called light-space. Shadows are cast where a surface is obstructed from the light emitter. In other words,

wherever shadows map to, there is an object with a depth higher than another object, from the perspective of the light. To be able to calculate this, the view matrix, instead of where the camera representing the eye of the user looks at, has to be constructed along the normal vector of the light. The projection matrix depends on the type of the light. In the case of a point light (it emits light in every direction equally, similar to an infinitely small light bulb) perspective projection is generally used. In case of directional lights (where the light rays are practically parallel, i.e. light from the sun) orthographic projection is used. The radiosity algorithm dictates that a patch distributes light in every direction that is visible from it, so perspective projection is the more fitting one.

In order to be able to construct shadow maps two passes are needed. First a framebuffer with only a depth map (called the shadow map) is created. The scene is rendered into this depth map from the viewpoint of the light. In the second pass this depth map is bound as a read texture. The scene is rendered normally. However, in the vertex shader a light-space transformed four-component vector position variable is also output. In the fragment shader it is necessary to divide the xyz components of this vector with the w component. This is called the perspective divide and the graphics card automatically does this in the case of the gl_Position variable. From this stage it is possible to retrieve the depth value of every given fragment from the viewpoint of the light. The depth map is also available, which contains the closest position to the light at every fragment. The next step is incredibly simple: compare the depth (z) component of the light-space transformed position of the currently shaded fragment to the depth value found in the depth texture. If the depth is equal or less, then the fragment is not in shadow. Otherwise it is in shadow.

This approach is not perfect. Due to the limited resolution of the shadow map the edges of the shadow can be jagged. This can be improved with percentage-closer filtering (PCF) and a technique called cascaded shadow maps. Due to the limited accuracy of the buffer it is also possible that fragments that should be visible return a “non-visible” result after the depth map check, resulting in artifacts called “shadow acne”. This can be improved upon by introducing a small bias. The shadow mapping algorithm used in this project is kept rather simple, with only a small bias introduced, to eliminate shadow acne and percentage-closer filtering is used to smooth the shadow edges.

Technique 3 - Lightmaps: Despite their names being similar with shadow maps there are not many other similarities between the two. The definition of “lightmaps” is not a well-defined

term either, since it can refer to multiple different things in different contexts. Commonly it is used to refer to lighting information pre-calculated in modelling applications. Here, however, it is not used in that sense. Global illumination algorithms have to store lighting data for every surface since that is how reflected, indirect light information can be efficiently accumulated. Compared to local illumination models this ramps up the complexity significantly, since the lightmap write has to be handled on the GPU for the purposes of efficiency. Information about doing it with modern OpenGL is scarce, however. In this project lightmaps are stored per mesh. Their purpose is to store both the incoming light/radiance (irradiance) and the outgoing light/radiance values. They are mapped to the geometry according to the UV coordinates of the individual meshes. This unfortunately introduces a constraint where no mesh in the scene can have UV overlapped triangles. However, this constraint is not uncommon among lightmapping software, Unreal Engine 4 seems to suffer from this as well [32].

Discounting this limitation, lightmaps are an optimal way to store lighting information for the whole scene since they provide an easy way to make use of the accumulated lighting information. After the lighting update is finished the only necessary action to be taken is to multiply the diffuse value of the texture where a fragment maps to by the value stored in the corresponding lightmap.

Technique 4 - Texture-space rendering: Let us introduce the final coordinate system used in the project: texture-space. In order to be able to write to lightmaps there is a need for a coordinate system that contains every texel within the lightmap. If this system is used the fragment shader can visit and update all of these texels. If there is no UV overlap, it can be achieved by setting the gl_Position variable to the UV coordinates of every vertex. The end result of this operation is a texture that contains information at every texel where the mesh itself maps to. For radiosity this can be used to create a texture-space geometry buffer (G-buffer – similar to the buffer used in deferred rendering systems) which can be read back to the CPU and used to determine the information needed to select the next shooter. Aside from this, texture-space rendering can naturally also be used to update the lightmaps themselves. The reasoning behind this is that the lightmaps map to every UV coordinate of a mesh, thus if these coordinates are used for rendering the end result is guaranteed to 1-1 map lighting information to geometry, as long as the UV coordinates are correctly assigned.

Technique 5 - Z-buffer sort and mipmapping: The underlying idea behind using the GPU for sorting is incredibly simple: first retrieve or calculate a value for every element to be

sorted that represents where it should go within the list of ordered elements. After this, map this value to the floating-point range of [0-1]. Take the reciprocal of this value and use it as the Z-component of the rendered fragment.

In more concrete terms, in the project this technique is used to select the shooter mesh. Based loosely on [33], the shooter mesh is the mesh that stores the most energy. First create a framebuffer with a 1x1 texture as its render target. Define the vertices for a screen-aligned quadrilateral. This ensures that every shaded fragment is going to fully map onto the 1x1 texture. Next render this quadrilateral and the texture representing the average radiance energy of every mesh onto the quadrilateral.

Before the algorithm is continued it is worth taking a look at how the average radiance energy is obtained. OpenGL supports a texture downscaling method called "mipmapping". Normally mipmapping is used to scale down textures that are only displayed far away from the viewport, in order to save GPU memory. The reason behind this is that if something is far away from the camera the person observing the scene is not going to be able to notice a higher resolution texture versus a lower resolution one.

Mipmapping works by averaging the values of neighbouring texture pixels to output the new, lower resolution texels. In the case of a power-of-two texture it is possible to command OpenGL to generate mipmaps that are progressively downscaled by half. This creates a "mipmap pyramid". The top of the pyramid is going to have a resolution of 1x1. Because of the value averaging nature of mipmapping this 1x1 texture contains the average colour value stored within the whole texture. In the case of a lightmap texture (for example, the radiance texture) this value represents the average energy within the texture and thus it can be used to decide where the mesh that "owns" this texture goes within the list of meshes that have to be sorted by the amount of radiant energy they have.

The highest, 1x1 level of the mipmap pyramid can be retrieved within the fragment shader. As mentioned previously, this value can be used as the depth value of this fragment. Fortunately, it is possible to set the depth value within the fragment shader. Unfortunately, it still has to be mapped to the [0-1] range. [33] recommends multiplying the value retrieved from the texture by the size of the texture (getting the overall energy stored in the texture in question) and using the reciprocal of this as the fragment depth. This is a viable approach, however, due to the limited precision of the depth buffer and the possible sizes of textures within this project

there have been some concerns about floating-point inaccuracy. Thus, the proposed solution is to take the value from the top of the mipmap pyramid, add 1 to it and take its reciprocal. It is necessary to add 1 to it to avoid a division by 0 (if the value from the mipmap pyramid is 0) and to have final values that map to [0-1]. For example, the reciprocal of 0.01 is 100 but the reciprocal of 1.01 is 0.99. This mapping also follows the rule that a lower depth value is bigger than a higher one. Take, for example, 0.5 and 0.1 as average energy values from the lightmap. The algorithm should, of course, select 0.5. The reciprocal of 1+0.5 is 0.67. The reciprocal of 1+0.1 is 0.91. The depth buffer would allow the fragment with depth 0.5 to write and thus the mapping is correct.

Two questions still remain: first is, what value should be written for the fragment with the highest energy? The answer is, it should be a colour-coded equivalent of the mesh ID of the fragment. Colour coding is a CPU technique and thus it is explored later in the report.

The second question is: how can this value be retrieved and used on the CPU? Let us take a look at this problem.

Technique 6 - Texture readback: GPUs have a RAM-like memory to themselves where data can be stored, just like on the RAM of the CPU. Textures declared within OpenGL reside in the memory of the GPU, which cannot be accessed by the CPU by default. There are situations, however, when data calculated by the GPU is useful for the CPU as well. The reason for this can be to make further calculations or to simply store said data.

OpenGL provides a function that does exactly that, glReadPixels(). First enough memory has to be allocated on the CPU side for this function to work correctly. Afterwards, however, this function copies the values stored within a texture on the GPU to the memory allocated on the CPU. By default, this operation is slow, since it has to wait for every rendering operation on the GPU to finish and it also has to transfer data on the bus between the CPU and the GPU. OpenGL also has pixel buffer objects (PBOs) which can provide a speed-up, since these make the read-back operation asynchronous. However, due to their asynchronous nature, the speed-up can be incredibly situational.

Technique 7 - Multisampling: By default the rasteriser stage only produces a fragment if there is a triangle that maps to the centre of a given pixel. Afterwards the fragment shader is invoked for this pixel. If a triangle only covers a part of the pixel but not the centre then the pixel is left at its default value. In normal rendering this can lead to jagged edges when the

scene is displayed. Multisampling solves this issue by creating sample points within a pixel. If a triangle touches any of these the fragment is created and shaded. The fragment shader is only invoked once but only samples covered by the triangle are written to, the rest are left with their previous values. This operation is going to result in a texture that has more information than regular textures so when rendering to a framebuffer this texture has to be ”resolved”, which means downsampled. OpenGL has a built-in ”blit” operation that averages the values at the samples within a pixel and creates a normal texture. Custom, manual resolve operations can also be written.

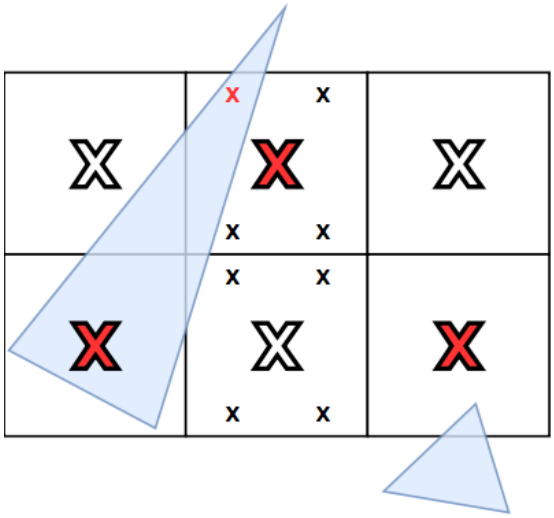


Figure 5.1: How different rasterisation modes work. A red middle ”X” means an invocation of the fragment shader. The left column shows normal rasterisation. The middle column shows rasterisation with 4x multisampling. The right column shows conservative rasterisation, which is not used in the project but is briefly mentioned in the Implementation and Future Work chapters.

Chapter 6

Implementation and evaluation

6.1 Basic OpenGL setup

Before getting to the complex lighting calculations the libraries used for the project have to be built and included into the project. Afterwards, a basic "skeleton" program has to be created, to handle drawing to the screen. This is usually done in the "render loop". This loop is not more than a while loop that runs until the user decides to close the program. This loop contains the logic, calculations and draw commands that are needed for a single frame to be drawn. These can be incredibly resource intensive and depending on the speed/complexity of these calculations and the available system resources/hardware it might go below what human eyes recognise as a continuous video. The limit for this is 24 loop executions (frames produced) per second (also known as FPS – frames per second). Of course, the higher this is, the smoother the video is perceived to be. On a 60Hz refresh rate monitor 60 FPS is considered to be a completely smooth video. (For a full discussion about this subject see [34].) Anything above this might result in an artifact called "screen tearing" – one part of the screen already shows the next frame while the other part of the screen lags behind. To eliminate this a technique called "vertical synchronisation" or simply "vsync" was invented, which locks the maximum possible framerate to the refresh rate of the monitor. Framerates are more important for games rather than global illumination renderers but since every renderer deals with frames in one way or another it is still useful knowledge.

As stated previously OpenGL uses shader programs. These are the small pieces of code that run on the GPU. Unlike DirectX11 with its own shading language (HLSL – High Level Shading Language), OpenGL does not provide built-in functionality for loading GLSL files. This functionality is implemented in the ShaderLoader class. It reads 2 files whenever it is instantiated: a vertex shader and a fragment shader file. It compiles these files into a shader program and displays if any compilation errors occur during compilation. These shader programs can pass uniform variables to their vertex/fragment shaders. Due to OpenGL's state machine nature these shader programs have to be bound before they can be used. This class also provides this functionality.

The model loader used for the project (Assimp) supports multiple model files formats but it is used to load Wavefront .obj files only. It is a rather simple file format that is human-understandable and can be edited by hand, if necessary. Wavefront .obj files can have material (.mtl) files that give further information about the object – for example the textures each mesh has. Assimp is a rather heavyweight library. The relevant things to know about it are that it considers the whole content of a .obj file as a Model and smaller sub-parts of the scene as Meshes. It also grants access to faces (triangles) and per-vertex data. There is a corresponding struct/class for all of these, however, the faces are generated manually from the vertex data in the main renderer.

In the implementation the ObjectModel class configures Assimp and initiates and conducts the object loading. ObjectMesh is mostly responsible for storing the relevant Triangle and Vertex data and handling draw commands – including binding the loaded textures of the mesh. Since this renderer deals with radiosity and radiosity only deals with diffuse reflections, only diffuse textures are used.

The project also has a Camera class. This deals with translating the mouse movements and the directional movement commands originating from the keyboard and constructs a lookAt matrix every frame from the information it currently holds.

Aside from a simple fragment program and a way to keep track of the user-defined lights (one way is explained later in the report) this information is enough to write a simple renderer that can load a scene and allow the user to observe it, rendered with Phong shading. It also describes the renderer "base" part of the project and thus from now on the focus is on the radiosity implementation itself.

6.2 Radiosity implementation

6.2.1 The basics

From the previous description of radiosity it can be seen that, in theory, it is not an extremely complicated technique. It boils down to dividing the scene into patches, storing their incoming and outgoing energy values, choosing the patch that has the highest outgoing energy and calculating the form factor between this patch and every other patch in the scene. Finally, the scene only has to be rendered with the resulting energy values. After this, the whole iteration can repeat until desired.

As simple as it is in theory, it does raise a significant amount of questions when it comes to the implementation. Among those, the 5 main questions are:

- How to represent patches?
- How to choose the shooting patch?
- How to find form factors? This has two sub-questions:
  - How to find the geometric factor?
  - How to find the visibility factor?
- How to distribute energy from the shooting patch to the other visible patches?
- How to find the final surface colours so they can be drawn on the screen?

Some of these questions have to be split into smaller parts for the implementation but this list gives a general outline of the problems that have to be solved. The rest of the Implementation section is going to give at least one solution for each of these questions. However, due to the many different, sometimes conflicting and generally incomplete implementation techniques published over the years it is worth investigating multiple possible answers for a given question. For that purpose, after a short "general plan" section a 1st, simpler implementation is given and evaluated, followed by a more complicated but in nearly every way better one. Without further delay, let us look at the general plan.

6.2.2 General plan

The plan draws inspiration from [33] and [35]. Neither of those provide full implementation details and thus as far as can be told the implementation described here is ultimately different, even if there are some shared techniques.

A radiosity renderer has to store accumulated illumination data for the whole scene. As already established textures are the way to make GPUs process and store data and thus those are going to be used to store illumination data. These are the so called "lightmaps". These lightmaps are stored for every mesh, according to the UV coordinates given to the respective meshes. The UV coordinates of meshes are often assigned by artists, working in modelling programs. Depending on how the original mesh textures are drawn up (and the workflow of the artist/company) it is possible that there are overlapping UV coordinates in order to save on the amount of data that has to be stored within the texture. Lightmap coordinates, however, have to be unique. There are standalone programs that handle a "UV-unwrapping" operation that automatically generates UV coordinates for a lightmap. After some investigation, this lightmap generation (also called mesh parameterisation) appears to be a vast and complicated topic that could be at the centre of a research project in itself. Because of this the current renderer only focuses on models with per-mesh non-overlapping UV coordinates. UV-unwrapping is once again visited in the "Future work" section, however.

Instead of just one lightmap per mesh radiosity requires 2 lightmaps. One lightmap needs to store the incoming light (also called incoming radiance – from this point referred to as irradiance) and another lightmap that stores the reflected light (or reflected radiance – simply referred to as radiance). Another advantage of using textures for light data storage (and thus texels as patches) is that it decouples the radiosity algorithm from the geometry. In other words, classical radiosity algorithms have to subdivide ("tessellate") the scene geometry into smaller patches if the quality of the radiosity solution is desired to be increased, which can be a complicated task. In the case of this renderer for an increased resolution the only thing that needs changing is an integer value.

Progressive radiosity can achieve the fastest results if the shooter is selected to be the patch with the highest radiant energy. As seen in the naïve implementation this can be impractical at times and thus an alternative solution is proposed. In brief, the second solution chooses the mesh that has the highest average energy and chooses the actual shooter accordingly. There

are other solutions that might warrant investigation in the future (for example: stochastic radiosity) but for the scope of the project the highest-energy-first progressive radiosity was deemed the most appropriate.

The shooter selection algorithm has to retrieve some information about the shooter. The shooter is treated similarly to a point light with one half cut off, in order to prevent it from distributing light behind itself. A point light like this would need information about its world-space position, its world-space normal vector and the per-RGB component intensity of the shot light.

The geometric part of the form factor is rather simple, if every patch within a scene is considered to be the same size:

$$dF_{dA_x \to dA_y} = \frac{\cos\theta_x \cos\theta_y}{r_{xy}^2}$$

For an explanation see the more complicated formula in Chapter 3. This version of the formula was adapted from page 40 of [35].

If the sizes are not uniform the area of the receiving patch also has to be considered. Here it is also worth mentioning that the energy that a patch shoots has to be scaled down to the size of the patch relative to the mesh.

The visibility factor is the harder part. According to [1] it can be calculated where a patch shoots its energy if a hemisphere is placed on the patch. According to [21] a hemisphere can be approximated with a hemicube. Both of these are examined in the further sections. Unlike many older radiosity techniques the hemisphere/hemicube is only used for the visibility factor and is not integrated over explicitly. This makes the visibility factor calculation similar to a more complicated form of shadow mapping.

The energy distribution happens by rendering every lightmap texel and adding the newly, per-fragment calculated irradiance and radiance (i.e. delta irradiance and delta radiance) values to the old, already stored lightmaps. The delta irradiance would simply be the shot light multiplied by the form factor, the delta radiance is similar but it also has to be multiplied by the value retrieved from the underlying diffuse texture of the fragment.

To get the final surface colours the scene has to be rendered normally but the diffuse colour of a fragment has to be multiplied by the irradiance value of a fragment. It has to be the irradiance value since the final render represents what the camera can see. The diffuse value of the surface might absorb some light and thus the incoming light has to be taken into account.

That is the high-level overview of the algorithm. The next sections present the implementation details.

6.2.3 Naïve implementation

It is worth noting that a data structure used in C++ is called "vector". This is different compared to a geometric vector used in OpenGL since the C++ vector is a dynamically allocated array. In order to avoid confusion the C++ vector is going to be referred to as "std::vector" (since it is part of the C++ standard libraries) and the geometric vectors are going to be simply referred to as "vector".

The algorithm presented uses the CPU for coordination and for information storage while the GPU does every calculation. In order to be able to do this, the irradiance and radiance maps have to be stored as two std::vectors per mesh and only transferred to the GPU when necessary. Every mesh is initialised with blank lightmaps, aside from lamps. The program is only designed to work with user-specified lamps but during mesh initialisation arbitrary meshes can be defined as lamps, if desired.

Every lamp needs a location, which is specified by the user. The program handles this by storing the order in which lamps are added to the scene and storing the location of every lamp added within an std::vector.

The shooter selection part needs to retrieve a few key pieces of information about the shooter (world-space position, world-space normal, shot light intensity). Those pieces of information plus the UV coordinates are rendered in texture-space per mesh and then read back into an std::vector each. Additionally, a unique ID is given to every triangle during the model loading stage. This is rendered and read back in the same way. This is a "preprocess" step within the algorithm. It does not directly contribute to the algorithm but it provides a way to efficiently retrieve information about every possible patch within the scene, which is useful, since all of them can end up being the selected shooter. It is not entirely dissimilar to the G-buffer used

in deferred renderers. This step requires the most GPU-to-CPU readback within the whole algorithm and thus at higher lightmap resolutions can take seconds. However, unlike most steps in the algorithm this step only has to be done once, at the start, so the time required is not deemed unacceptable.

The shooting patch selection in the naïve version is incredibly simple. Loop through the radiance information of all the patches on the CPU, calculate the luminance of each: [35] gave luminance as:

$$0.21 * redComponent + 0.39 * greenComponent + 0.4 * blueComponent$$

which is roughly equivalent to the per-intensity CIE XYZ tristimulus value averages which are given in [36]:

$$X = 0.49I1 + 0.31I2 + 0.20I3$$

$$Y = 0.17697I1 + 0.81240I2 + 0.01063I3$$

$$Z = 0.00I1 + 0.01I2 + 0.99I3$$

then choose the patch that has the highest luminance. Keep track of the index of this patch, its radiance value then retrieve the other data about it from the other stored std::vectors that is needed for the rest of the algorithm.

After this step the shot radiance is scaled down to size. For now, it is assumed that every mesh has the same number of patches and every patch is uniform size. In other words, the radiant energy of the shooter is divided by the number of texels the lightmap has, which is equal to the square of the size of the lightmap.

The geometric factor of the form factor between two elements is calculated according to the previously stated geometric factor formula (adapted from [35] but [33] also gives a similar one) and happens in the lightmap update render pass. It is influenced by the distance between the two elements and their orientation. This formula can be used during the lightmap update part of the algorithm, since during that phase every mesh is rendered in texture-space. From that it can be seen that every patch is rasterised into a fragment and thus the fragment shader is invoked as many times as the number of patches in our scene. These fragments are the "receiver" patches, because they receive energy from the shooting patch. In the fragment shader it is possible to get every interpolated attribute of the receiver that is needed for the

geometric factor: its world-space position and its world-space normal. The required attributes of the shooter are passed as uniform variables. From the world-space positions of both the shooter and the receiver their distance can be calculated and from their world-space normal vectors, their relative orientation. Since this happens in every fragment shader the geometric factor is calculated for every patch.

The visibility factor calculation is done according to [33]. Instead of a traditional shadow mapping algorithm a hemispherical projection is used. Instead of using the Z-buffer for depth testing, every triangle is given a unique ID then rendered into a framebuffer, which can also be called the "item buffer".

The way the triangle IDs are assigned is worth examining: every triangle can be given a 3-component (RGB) ID. In order to support a high amount of triangles it is worth finding a way to use the RGB components efficiently. In the end an "overflow counter" based approach was chosen. Keep a static integer counter and increment this whenever a triangle is instantiated. (The static counter should start at 1 so 0 can be used to signal that no triangles map to a given texel.) Simply assign the counter to the first component as long as it does not go above an arbitrary number. In order to avoid precision issues 255 was chosen to be that number. Once the counter exceeds that, set the first component to 0 and the second component to 1. If the first component exceeds 255 again then set it to 0 and set the second to 2. Likewise with the second and third components. Continue this process until every triangle is processed.

The triangles are rendered (with their ID given as their flat shaded colour – for this the triangle ID has to be passed to every vertex of a triangle as a vec3, i.e. a 3 element floating point vector attribute) into a colour texture attached to the previously mentioned framebuffer. Subsequently the scene is rendered normally, however, not from the viewpoint of the camera but from the viewpoint of the shooting patch. For this to be possible the lookAt matrix has to be constructed from the world-space position of the shooter and along the world-space normal of the shooter. A specific projection matrix is not constructed before the shader run, the hemispherical projection is done in the vertex shader for the item buffer render pass. The fragment shader sets the colour of the output fragment to the ID colour of the triangle.

With this the item buffer is constructed and held in a texture. This texture is passed to the lightmap update vertex shader along with the lookAt matrix from the viewpoint of the patch. In the vertex shader an output variable is held that transforms the vertex positions

to camera-space, using the lookAt matrix of the shooting patch. In the fragment shader the camera-space position is projected to a hemisphere, scaled up to the [0-1] range for texture lookup and the corresponding colour in the item buffer is retrieved. If this matches the colour ID of the current fragment then the fragment is visible else it is not. Return 1 or 0 accordingly.

To get the whole form factor the geometric factor hat to be multiplied by the visibility factor which is multiplied by the shooter radiance (which is passed as a uniform). This operation is equivalent to "implicitly" integrating over the hemisphere, since energy is distributed to every patch that projected to the hemisphere. The end result of this calculation is the newly received irradiance value: the delta irradiance. To get the delta radiance the delta irradiance has to be multiplied by the diffuse value of the fragment (retrieved from the diffuse texture of the fragment). These delta values are then added to the already accumulated irradiance and radiance values. Retrieve the values of these from their corresponding lightmaps and add the respective delta values to them. Using multiple render targets both the new irradiance and radiance maps can be output in one pass. After the render pass this data is read back to the CPU and stored in two std::vectors, updating the accumulated lightmap data.

This concludes the first radiosity algorithm. The iteration part simply loops through this algorithm as many times as desired.

The way one iteration is defined in radiosity algorithms is not always clear. In this implementation the iteration starts with the shooter selection and ends when that shooter has shot its energy over all the receiving patches in the scene.

To display the scene with the calculated radiosity values textures have to be created out of the stored lightmap data. These are bound then the scene is rendered normally, from the viewpoint of the camera. In the fragment shader the diffuse value of the fragment is multiplied by the irradiance value of the fragment. This value is the output of the fragment shader run.

6.2.4 Naïve implementation results and evaluation

This is a functional but sub-optimal radiosity implementation. It exhibits both soft shadows and colour bleeding. However, the number of artifacts is also significant. One of the issues is that the hemispherical projection used in the visibility calculation requires curved edges (in order to successfully project triangle edges to a hemisphere) but the OpenGL rasteriser only

outputs straight edges [33]. One way to lessen the impact of this is that if the scene is highly subdivided, i.e. it contains a high number of triangles. Rendering many triangles is expensive so if this is not otherwise necessary (for example: it might make sense to use many triangles for complicated meshes but for a straight, flat wall a low amount is usually desirable) then it only wastes processing power. This step can be done on the GPU with tessellation but that is no less wasteful. Additionally, the radiosity implementation presented here is supposed to be decoupled from the geometry and thus a tessellation stage would negate the advantages provided by the patchless algorithm.

The second problem is that the geometric factor calculation presented in GPU Gems is not entirely correct, at least not the way it was used in this implementation. The problem with it can be described this way: regardless of the visibility factor, the far sides of meshes (compared to the location of the shooter) should have a form factor of 0. The computed geometric factor, however, was above 0, leading to illuminated edges of faces that did not face the light emitter. This proved to be a hard to track down bug and it was present even in the advanced implementation for a while.

The last problem is that it is incredibly slow because of the CPU shooter selection. Since the shooter is chosen to be the texel that has the absolute highest radiance energy in the scene, the CPU has to loop through all of the texels within the scene during every iteration. It gives it a time complexity of $O(N^2 * M)$ where N is the resolution of the lightmap and M is the number of meshes in the scene.

The lookAt vector calculation is also slightly faulty in this implementation, which can lead to some patches not shooting their energy.

Due to its speed no full scenes were rendered with this implementation. However, a few screenshots were taken to show off the artifacts and colour bleeding:

Figure 6.1: A significant amount of artifacts can be seen if a hemisphere is used for the visibility calculation. The framebuffer print GPU debugging technique can also be seen. The framebuffer contains the ”item buffer”.

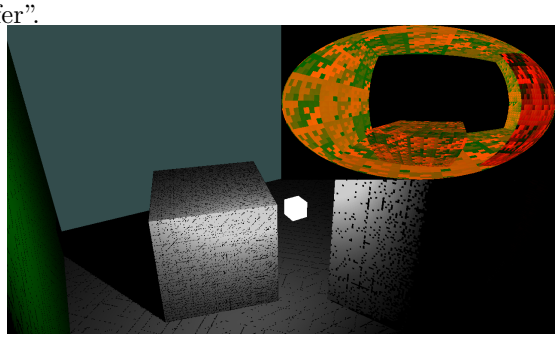
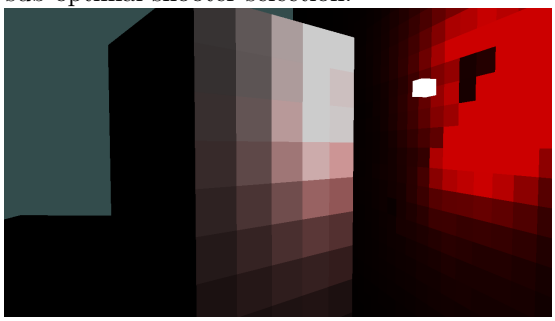


Figure 6.2: Colour bleeding with artifacts. Note that the amount of colour bleeding might be exaggerated due to the sub-optimal shooter selection.



The advanced algorithm presents a way that solves these problems. Radiosity and global illumination in general are incredibly deep topics, however, so even that algorithm has a few problems in its current state. In the future work section a few possible ideas to solve those problems are given.

6.2.5 Advanced implementation

Let us start by improving the performance. It can be seen that the original shooter selection spends a significant amount of time repeating computations it has already done in previous iterations and also iterating over texels that do not map to a mesh. This is possible because not every mesh fills out the whole UV space. Therefore, the first possible improvement is removing texels from the computation that can be neither shooters nor receivers. The best place to do this is during the preprocess stage. For this the texture-space triangle ID data is used. If the sums of the individual RGB components of the retrieved ID are above 0 it can be

known that a triangle is present. In this case add the index of this texel to an std::vector that stores valid texel locations per mesh.

This step removes the invalid texels from the operation. However, at this point the shooter selection is still a CPU operation which heavily deteriorates speed. In order to be able to map this algorithm to the GPU the highest-luminance-texel-first rule has to be slightly relaxed. Instead of finding that texel, the mesh with the highest average radiant luminance is chosen. This slightly modifies the previous definition of an iteration. From now one iteration is considered to start when the shooter mesh is selected and it is finished once every patch has shot its energy within the mesh. The shooting itself becomes a "sub-iteration". It is not guaranteed that the patch with the highest energy immediately shoots. However, it is assumed that the mesh that holds the highest amount of average energy does contribute the most to the lighting in the scene.

The implementation relies on the fact that lightmaps are stored per mesh. This makes it possible to use the Z-buffer sort technique described earlier in the report: first every mesh is assigned a unique RGB ID the same way the triangle IDs were assigned but instead of using 255 only 100 is used as the "counter cut-off" to eliminate any possible depth precision issues and to make it easier to translate the RGB ID back into an integer ID. Afterwards the RGB ID is passed as a uniform variable for each mesh. The Z-buffer sort is then performed and the resulting ID is read back to the CPU.

At this point a peculiar "feature" of C++ is worth mentioning. The read back ID is a float variable between 0 and 1. To get the integer ID the red component has to be multiplied by 100, the green component by 100*100 and the blue component by 100*100*100 and then all of them summed up. The problem happens with a particular value: 0.59. If 0.59 is multiplied by 100 then cast to an integer the resulting value ends up being 58. The error itself is not hard to understand: due to the limited precision of floats C++ interprets 100*0.59 as $0.58\dot{9}$. A cast to an integer truncates the floating-point value and the result ends up being 58. The fact that it only happens with 0.59 and no other float with 2 decimal places makes this an incredibly hard to trace bug if not noticed and eliminated right after it is introduced. Fortunately the fix itself is easy: simply use std::round() before casting the value to int. This results in 0.59*100 being equal to 59.

The Z-buffer sort returns the ID of the shooter mesh. A separate shooter selection function is constructed. Unlike the previous shooter selection, this one only retrieves the index of the next shooter from the valid texel locations std::vector then uses this index to return the attributes of the shooter that the rest of the algorithm needs. It also zeroes out the radiant energy of the shooter.

The complexity of the shooter selection itself is thus $O(1)$ since no actual computations or searching happen, it only does array lookups which are constant time. The mesh selection requires one draw call for each mesh, which is $O(N)$ draw calls where N is the number of meshes. Mesh selection only has to happen once the chosen mesh has no more shooters to be processed, thus it happens relatively infrequently.

The second issue was the faulty geometric factor calculation. Instead of adapting the geometric factor code presented in [33], the one presented in [35] is used. Coombe in GPU Gems uses world-space calculations to get the geometric factor [33]. Szirmay-Kalos, however, calculates it in camera-space [35]. His geometric calculation fortunately does not exhibit the "illuminated far-wall edge" artifact. Less fortunately, however, the given algorithm shoots light backwards from a given patch. The fixed algorithm requires the "ytox" variable to be the negative of the camera-space position vector of the fragment, normalised. Similarly, the cos theta x has to be the dot product of the normal transformed into the light-space of the patch and positive "ytox". This solves the backwards light distribution problem and the incorrect far-wall illumination too.

That leaves only one major problem remaining: visibility calculation with hemicubes. This step involves heavy lookAt matrix manipulations so let us examine why a simple implementation would not always work.

In the Naïve implementation section an important detail was omitted about lookAt matrices. For a bit of a recap: a lookAt matrix requires a position (where the camera is placed within the scene), another position (the camera looks at this point within the world – if a camera is desired that looks in a particular direction from a particular position then the first position vector has to be added to the vector that describes the point the camera looks at – this is the case when checking for visibility from a particular patch) and an upVector, which defines the orientation of the camera. The upVector is generally desirable to be at a right angle to the direction vector of the camera. The problem is, in 3D space there are an infinite amount of vectors that are at

a right angle to another. One, commonly used solution is to define a “worldUp” vector which is usually (0, 1, 0) – that is, it points towards positive Y. This can be considered as a global vector to which it is possible to compare every other vector within the scene. Finding the upVector for the camera placed on the patch then is a simple matter of first finding the right vector of the camera (which is done by taking the cross product of the normal vector of the shooter and the worldUp vector) then calculating the upVector itself by taking the cross product of the right vector and the normal vector of the shooter. This approach, however, does not always work. Fortunately, there are exactly two cases when it fails: when the normal vector of the shooter is parallel with the axis defined as the worldUp. If the worldUp is defined as (0, 1, 0) then normal vector cannot be (0, 1, 0) or (0, -1, 0). Therefore it is possible to conditionally modify the worldUp depending on the normal vector of the shooter. For example, if the normal vector is (0, 1, 0) then the worldUp can point towards negative Z (0, 0, -1) and if it is (0, -1, 0) the worldUp can be (0, 0, 1). This is how it is used in the program.

The previously given method only describes the visibility lookAt vector calculation for a very simple light. The hemicube is exponentially more complicated than that. First, it has to be known about the hemicube that it is – like the hemisphere is a sphere cut in half – a cube cut in half. The “full” face lies where the normal vector of the patch points to and the windows (or side faces) are the four faces that connect to the full face, just cut in half. For each of these faces a lookAt matrix has to be constructed. This is a non-trivial task and is also not particularly well covered in literature or online resources, so let us take a look at a method developed for this project and dubbed “double up”.

The method involves defining a worldUp vector similarly as it was described previously. After this a right vector and the upVector are calculated for the shooting patch. However, these are not directly used and instead considered as the right vector and the upVector for the hemicube. Now there are two up vectors, one for the world, the other for the hemicube. Hence the ”double up” name. For the front, left and right faces the hemicubeUp can be directly used as the upVector and the left face looks along the negative hemicubeRight vector, the right face looks along the positive hemicubeRight and the front face simply looks along the normal vector of the patch.

The situation with the up and down faces is slightly more tricky. Since a hemicube is in use and the faces are at right angles to each other it can be seen that the upVector of the up face can be the negative normal vector of the patch. Likewise, the upVector of the down face can

be the positive normal vector of the patch. The up and down faces look along the hemicubeUp and negative hemicubeUp vectors, respectively.

That gives a solid description of the way the lookAt matrices are constructed. Let us examine the rest of the algorithm.

Overall, the things that have to be done are similar to a more complicated form of shadow mapping. Shadows from point light sources can be calculated using a 3D (cube) depth map. These depth maps always require 6 textures to be bound – one for each face. However, since the hemicube only has 5 faces binding another texture would be a slight waste of resources, so here it is done with 5 2D textures.

In order to perfectly fit together the faces of the hemicube a 1:1 aspect ratio perspective projection matrix is used, with a 90° field-of-view. The scene is rendered 5 times, from the viewpoint of every face. The view matrices and depth textures are then passed to the lightmap update shader where the visibility test happens for every fragment. If the fragment is visible from one of the faces (using the normal Z-buffer depth test technique) then the visibility factor should be 1. Otherwise it has to be 0.

It can be noticed that what was actually rendered as the depth maps was a cube, without the back face. In order to create a true hemicube the depth map textures have to be initialised with maximum depth values on the side that is desired to be cut off. Fortunately, the geometric term prevents fragments to be illuminated that are behind the shooter, so this step is not necessary.

Since this procedure is based on traditional shadow mapping it does have a few issues. Shadow maps are finite and surfaces are continuous. This can lead to artifacts on surfaces that are at an angle from the shooter, called “shadow acne”. Due to the use of lightmaps the shadow acne manifested in barcode-looking lines even on visible surfaces in this project. A small bias eliminates this issue. The individual faces of the hemicube can also suffer from an error called oversampling. It simply means that a fragment maps to the outside area of a depth map. These fragments should be considered as not visible from a given depth map. In order to be able to do this, a border “colour” (in reality border depth) of maximum depth is assigned and the textures are set to use this whenever oversampling occurs. Special care should be taken with fragments that are exactly on the plane of the shooter yet are not visible from it (i.e. other patches on the same surface that the shooter occupies). These fragments map to a depth value of 0, so a conditional can set these to be marked as non-visible.

Next, let us revisit the form factor calculation. The previous form factor formula used assumed unit sizes for patches. This is not quite true in this renderer so let us examine how it can be improved. First, let us look at the full patch-to-patch form factor given in [16]:

$$dF_{dA_x \to dA_y} = \frac{\cos\theta_x \cos\theta_y}{\pi r_{xy}^2} V(x,y) dA_y$$

The only extra element that needs to be calculated is dA_y which is the area of the receiving patch. Calculating this is rather simple: individually find the surface areas of every mesh by calculating the areas of every triangle within the mesh then adding the values together. Next divide this value by the amount of texels a mesh maps to. This is the average texel area in that mesh. Afterwards insert this value into the old radiated energy and form factor calculations. The dividend, instead of being r_{xy}^2 is πr_{xy}^2 so multiply the old dividend by π. OpenGL by default does not define a unit/scale system so these changes would make a general-usage renderer (as opposed to, for example, a game engine at a company) too dark. The proposed fix is to define a "base" area of 1 unit squared for texels. This might not be strictly realistic but it allows patch sizes to influence the amount of radiated and received energy and fixes the darkness problem.

At this point black edges along the edges of faces are observable, which is not ideal. A uniform fix would need a UV-unwrapper so this fix only works where meshes have UV islands for every face. Black edges are caused by the rasteriser not producing fragments for edge texels during lightmap passes. This can be fixed with multisampling.

Unfortunately, this usage for multisampling requires a custom resolve pass, which seems to be a rather rarely used feature and thus it is not too well documented. A normal OpenGL resolve averages the sample values within a pixel, which can throw off texture-space calculations in case a pixel has samples that are not covered and are left at their original values. This is where a custom resolve pass comes in: draw a screen-aligned quadrilateral for every mesh, shade every pixel within the bound multisampled texture, loop through all the samples within every pixel and simply write the one with the maximum value. A fragment shader is only invoked once for a pixel no matter the sample amount so every covered sample has the same value, thus retrieving the maximum value is a valid strategy.

Additionally, centroid sampling is used for the multisampled passes. This means that attributes are not interpolated at the pixel centre but at the centre of the covered area of the pixel.

This way the interpolated values do not get ”extrapolated” i.e. they are correctly calculated within the covered area [37].

Finally, with this setup texture filtering for the lightmaps can be used with rather good results even on low lightmap resolutions. Texture filtering means that when the value is retrieved from a texture multiple nearby pixels contribute to the final value. This results in smoother lightmaps instead of the blocky ones that can be seen with low lightmap resolutions without texture filtering.

That concludes the radiosity part of the algorithm. A final adjustment can be done before rendering the scene, however. It is called gamma correction.

Most monitors exhibit the property that, if sent twice the power/voltage at a particular pixel location compared to a previous value the second pixel is not going to be twice as bright as the first one was. Instead, doubling the voltage results in the pixel being about as bright as the first value to the power of 2.2 [38]. This is called the gamma of the monitor. It is similar to how the human eye processes colours, however, it moves the colour output into a non-linear space. It is important to take this into account when rendering the final scene after all the lighting calculations are done since most of the intermittent calculations are done in linear colour space. Failing to apply gamma correction can lead to parts of the scene being unreasonably dark. Fortunately, gamma correction is rather easy to do: after all the lightmap calculations but just before drawing to the screen, the final output fragment colour has to be raised to the power of (1/2.2). Of course, 2.2 as the gamma value is not correct for every monitor. Depending on its production technology (CRT, LCD, OLED, etc.), the manufacturer of the monitor or the settings of the monitor the gamma might be different. Advanced renderers (commonly game engines) let the users specify the required gamma value. Having said that, this project sticks to the commonly used 2.2.

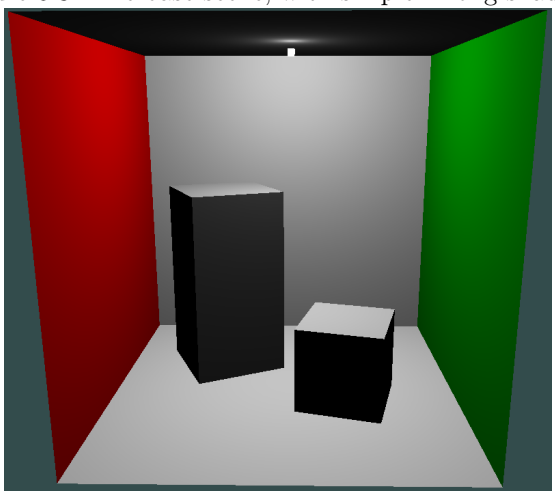
6.2.6 Advanced implementation results and evaluation

Simple lighting calculations can easily be performed even on aging graphics hardware. However, global illumination methods have to calculate and store lighting information for the whole scene, which makes them incredibly resource intensive. This is naturally true for radiosity as well. The project development machine is a laptop equipped with an Nvidia GeForce GTX 560M GPU and an Intel Core i7 2630QM CPU, with 12GB DDR3 RAM. In 2011, so about 7 years

before the time of writing these components were medium/high performance laptop-quality parts. Unfortunately, the computational power of this system is only a minuscule fraction of what a modern high-end system can provide. For this purpose, the software developed for this project is highly configurable: the quality of the radiosity solution (that is, the number of patches) can be configured per mesh and arbitrary scenes can be loaded (from simple to complex – as long as every mesh is completely UV mapped and has no overlapping UVs). Due to the meagre computational power of the test laptop the test scenes were kept relatively simple (based on a custom-made version of [39]) and the 3 lightmap sizes shown are either 32x32, 64x64 or 128x128. The first is fast but exhibits some artifacts, the third is slower but looks significantly better and the second is a compromise between the two.

Evaluating graphics applications is not an easy task. The best way would be to recreate the scene in real life and compare pictures produced by professional cameras with images produced by the renderer. Unfortunately, due to the lack of equipment this was not possible for this project. Comparing the renderer with other, similar renderers is also not entirely representative, since the current renderer lets users specify point(-like) light sources at runtime, instead of using one-sided light sources built into the geometry. Because of this the produced images are unlikely to be the same. Having said that, the appendix is going to contain links to a few other radiosity renderers so the readers can take a look at those. The renderer presented here loads the lamp itself from a Wavefront .obj file so the lamp model can be changed. However, this is not an explicitly planned functionality so it cannot be said how flexible the renderer is regarding different lamp models. Because of these reasons, it was decided that a Phong shaded image is going to serve as the base image for the evaluation and the radiosity-rendered images are going to be compared to it. This way it is possible to see how global illumination improves image quality. Performance notes are also given wherever appropriate.

Figure 6.3: The base scene, with simple Phong shading.



Unless otherwise noted, the following screenshots use all of the implemented improvements (for example multisampling) and the basic quadratic attenuation. The radiosity iteration was allowed to run until both coloured walls were allowed to shoot their accumulated energy. Also keep in mind that lights are placed by hand, so there might be minor illumination differences.

Figure 6.4: 32x32 lightmaps - 5522 shooting patches processed in 1min 47sec - 19ms/shooter.

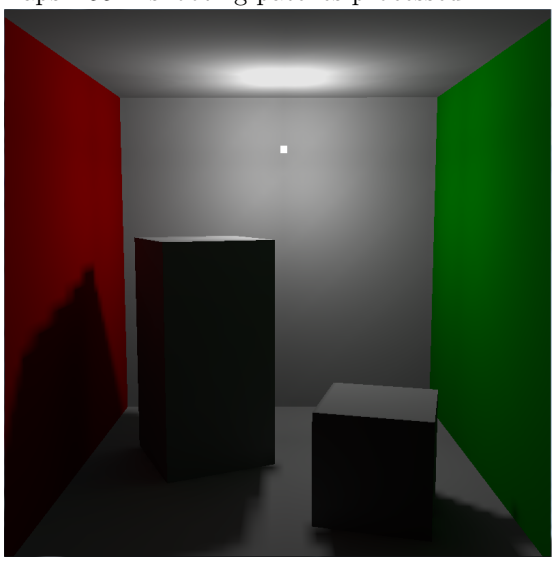


Figure 6.5: 64x64 lightmaps - 22043 shooting patches processed in 7min 57sec - 21ms/shooter.

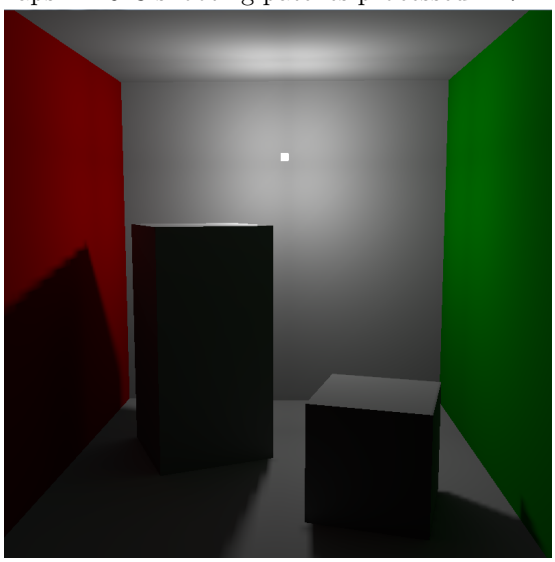
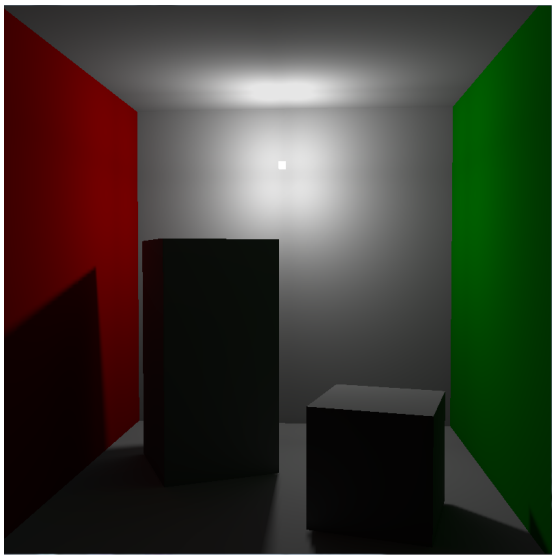


Figure 6.6: 128x128 lightmaps - 88152 shooting patches processed in 44min 16sec - 30ms/shooter.



The following 3 images are from the 128x128 render, showing off the scene from different viewpoints:

Figure 6.7: Here a ”shadowed shadow” can be seen - the shadow is lighter at some areas, darker at others.

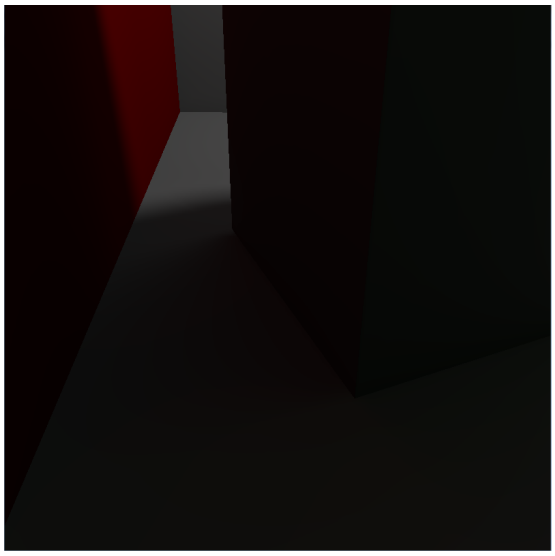


Figure 6.8: Faint green colour bleeding can be seen on the cube.

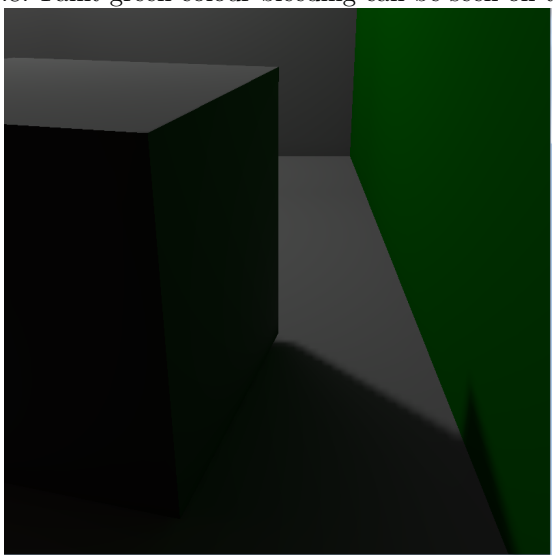
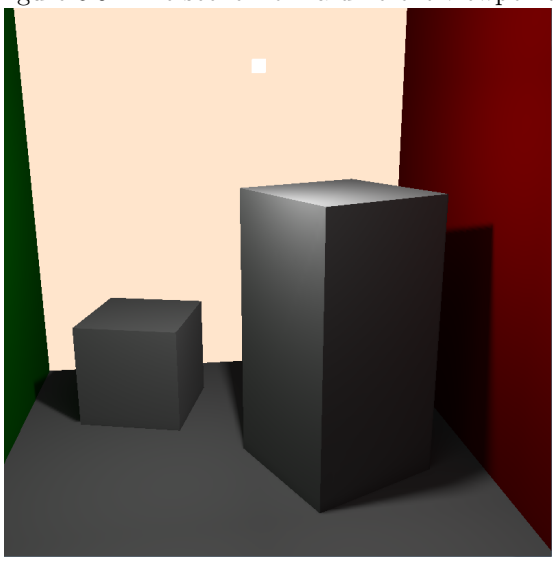
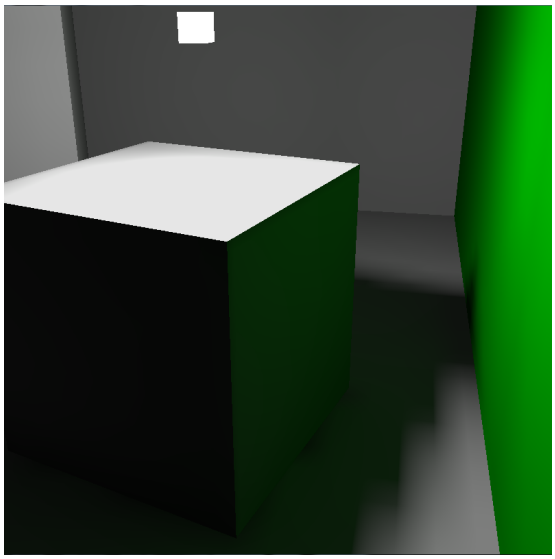


Figure 6.9: The scene from a different viewpoint.



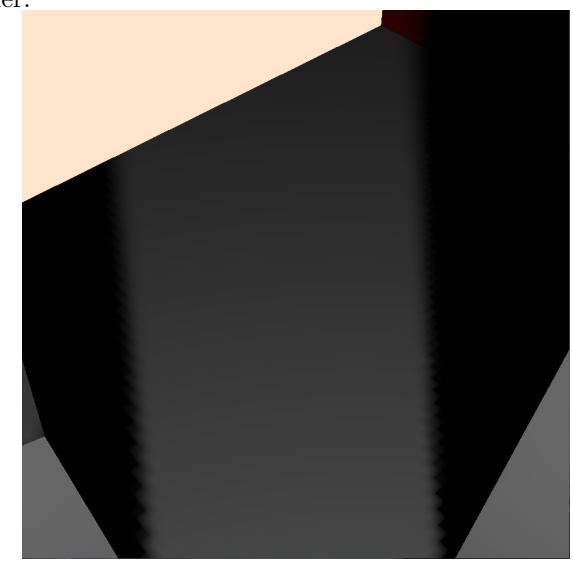
All of the previous images exhibit the typical radiosity properties: soft shadows, colour bleeding and view independence, the biggest difference is in the quality of the shadows. The previous scene only showed some faint colour bleeding, so let us look at a light placement that makes the effect stronger:

Figure 6.10: 32x32 render. Strong colour bleeding from the nearby wall due to the placement of the lamp.



Due to the fact that the renderer uses an area light the shadow edges also get smoother the farther a shadow is from the object that cast it:

Figure 6.11: 128x128 render but only the lamp is processed. Note how the shadow edges get smoother and smoother.



Let us also take a look at how multisampling and linear filtering improve the scene. All of the renders are with 32x32 lightmaps:

Figure 6.12: No multisampling nor texture filtering. The scene is blocky and artifacts can be observed.

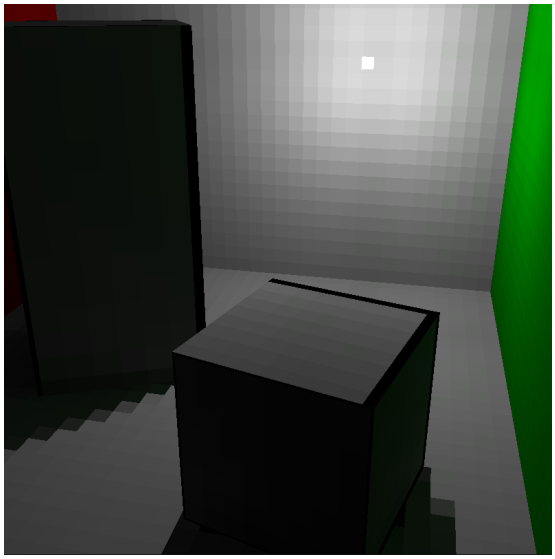


Figure 6.13: Multisampling gets rid of the artifacts.

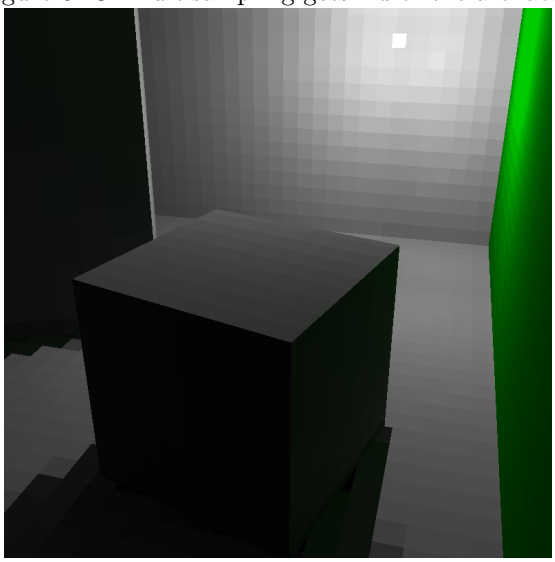
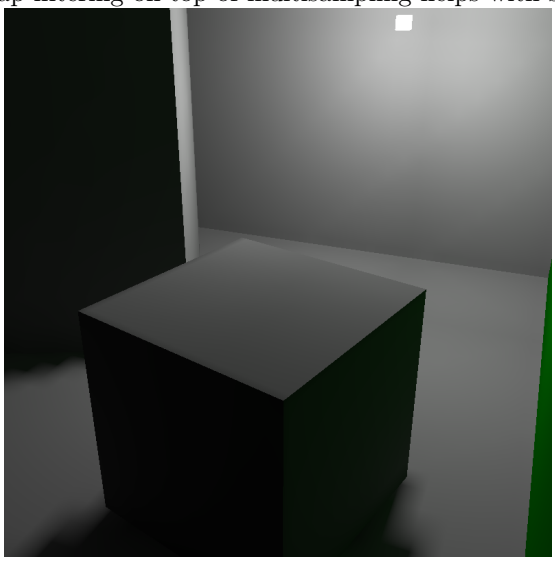


Figure 6.14: Lightmap filtering on top of multisampling helps with smoothing the scene.



Finally, a few screenshots from a scene with a more complicated light placement, 128x128 lightmaps:

Figure 6.15: The whole scene.

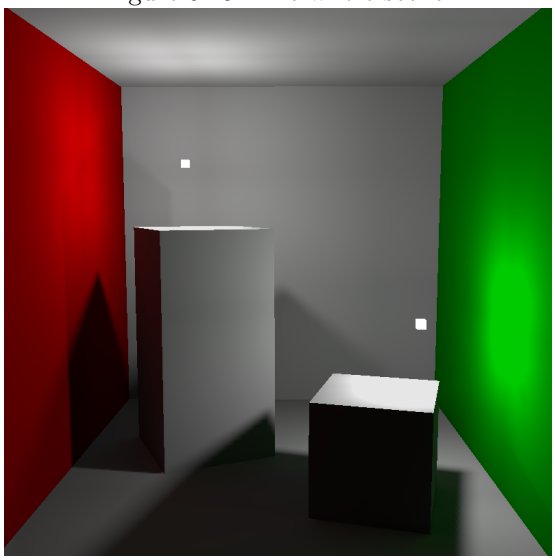


Figure 6.16: Complex, lighter and darker shadows.

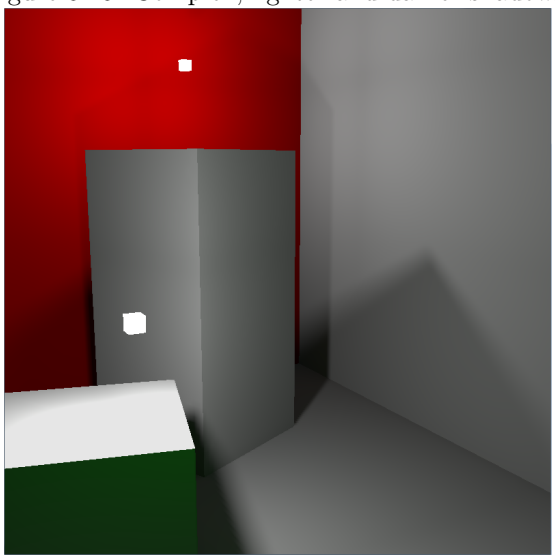


Figure 6.17: The cube partly blocks out the colour bleeding from the green wall.

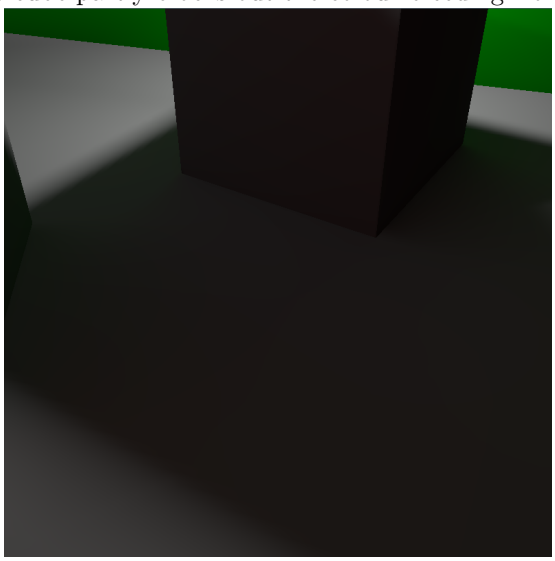
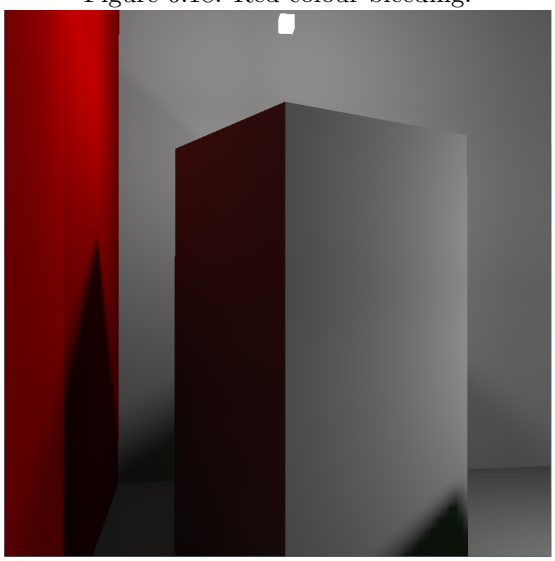


Figure 6.18: Red colour bleeding.



These are the more interesting settings combinations for the renderer. It has a few more functionalities (e.g. using linear attenuation instead of quadratic - this makes scenes lighter and colour bleeding more prominent) but for the sake of brevity they are not extensively visited here.

At this point it might be worth saying a word or two about automated tests and OpenGL. Test-driven development (TDD) and unit tests are important for large-scale projects. As ever, pure OpenGL renderers (i.e. no game logic, just focused on the rendered output) seem to be the exception to the rule, as it is hard to automatically test rendering output (generally requires a human observer) and is not considered to be worth the effort [40]. Because of that, unit testing is not part of the project at the moment. However, in the future (especially if the project expands) this issue might be revisited.

Overall, the program has all of the features promised in the Design and Specification chapter and to reiterate, the renderer part exhibits all of the prominent features of radiosity. Additionally, every computation is done on the GPU. Having said all that, the renderer is far from perfect. As the final images, let us take a look at a scene where the some of the meshes are hard to UV map and/or have UV coordinates that are not optimal for texture-space rendering:

Figure 6.19: 32x32 lightmaps, texture filtering off. Black spaces for non-rasterised triangles can be seen along the UV edge.

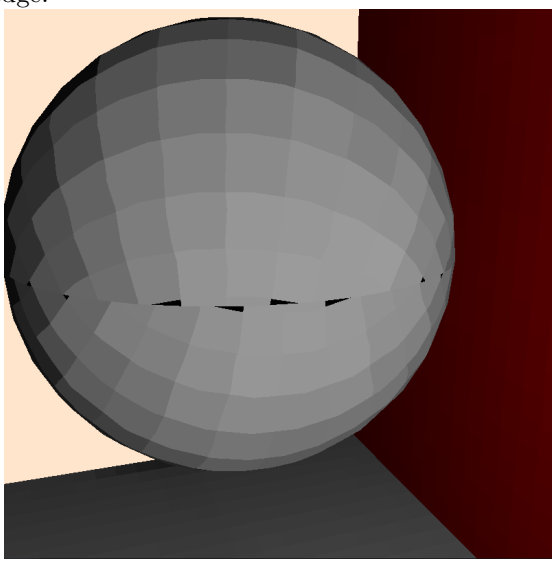
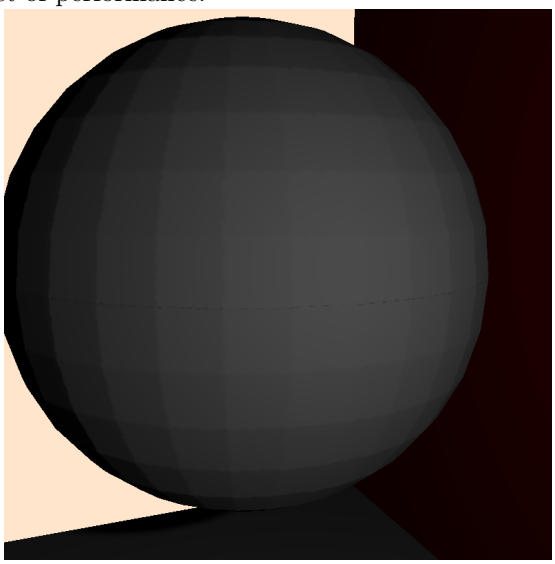


Figure 6.20: 512x512 lightmaps with texture filtering. The amount of artifacts can be reduced, however, at a heavy cost of performance.



The previous images show artifacts due to lightmap triangles not being rasterised, even with multisampling. This is due to the UV coordinates of the mesh not being optimal for the texture-space render passes. On modern hardware a conservative rasteriser (i.e. invoke every fragment shader whose pixel is touched by a triangle) could be used to reduce the impact of this problem. However, to fully fix this and other issues, there needs to be a way to create lightmap UVs automatically. This leads to the next chapter: the shortcomings of the current implementation and the work that needs to be done in the future.

Chapter 7

Shortcomings and future work

7.1 The requirement of non-overlapping UVs

By and far the biggest shortcoming of the program is that it requires non-overlapping UVs per mesh. It is by no means a unique limitation of the current program (see Unreal Engine 4, which does not tolerate overlapping UVs either [32]) but it does force users of the program to create meshes in a way that avoids overlapping UVs. Even then, sometimes those UVs are not completely ideal for texture-space rendering. The answer to this is a UV-unwrapper library. Creating an optimised UV-unwrapper is not a trivial task and is commonly delegated to other programs (DX UVAtlas [41], etc) and even modelling software. Therefore, it would make sense to write an OpenGL UV-unwrapping library that can create UV maps for meshes at runtime. This could integrate into the model loader used with this project (Assimp) or handle data with a specific format. It could either generate UV maps per mesh (in which case individual meshes would have more patches mapping to them and the current setup of the renderer would only require minuscule changes) or generate a lightmap for the whole scene. The latter is good for performance reasons: fewer textures are needed to be bound and it would allow the user to have a finer control over the overall number of patches.

There are a few issues that a UV-unwrapper has to handle. Performance is an important metric, since one of the main points to write a UV-unwrapper is to be able to handle large scenes. These scenes can contain hundreds of thousands of triangles. The problem is similar to bin packing, which is NP-hard [42] so an operation like this could take a long time. Another

challenge is to assign arbitrary triangles in such a way that as little as possible texture space is wasted while allowing texture filtering on the side of the renderer. This is possible by leaving a small "gutter" between triangles and before using the lightmap to draw the scene a small kernel program can be used to dilute the lightmap. This basically means giving every packed triangle a small "skirt" that lies out of their UV coordinates but inside the texels that linear filtering reaches. Aside from just multisampling, the use of a conservative rasteriser might be worth trying in conjunction with the UV-unwrapper for better lightmap quality.

7.2 Performance improvements

The current implementation reads back data from the GPU to the CPU every time a lightmap is updated. At higher lightmap resolutions (for example: 512x512) on the test PC this can take a significant amount of time (around 0.1s - 0.2s overall for updating the lightmap for every mesh in the test Cornell-box-like scene). The upside of this implementation is that it is memory efficient: the delta irradiance and delta radiance maps only have to be stored for the mesh being processed. However, in the future tests should be carried out on a range of diverse hardware to see if the performance problems persist. If the answer is yes, it would be possible to transform the algorithm in a way that would use multiple textures to keep track of the delta irradiance and delta radiance through a whole iteration. In this case, readbacks would only be have to done once per iteration, before a new shooter mesh is selected. This would normally cause problems with updating the scene in the middle of an iteration, however, the delta irradiance texture could be passed to the final rendering step and added to the irradiance map there.

Another possible way of optimising performance would be to move the whole algorithm to the GPU. This would eliminate the need to do any lightmap data transfer between the CPU and the GPU but moving the coordination from the CPU to the GPU would be a non-trivial task.

7.3 Adding full HDR

HDR stands for "High Dynamic Range". Normally screens accept RGB colours in the [0-1] float range, this is generally called "Low Dynamic Range". This limits the range of lighting a renderer can produce and results in light values that are not realistic. In order to alleviate

this, HDR rendering does not limit the amount of light that is incident on a surface or the light values that are shot from a light. At this point in time the renderer uses a limited form of HDR. Values that are contained within lightmaps are not clamped to the [0-1] range but for the final render the values are clamped to that range. Monitors cannot accept colours outside this range but normally "tone mapping" functions are used for this purpose. Some research would have to be conducted to find (or write) one that is suitable for radiosity.

7.4 Writing a detailed tutorial series about the implementation presented here

Due to the lack of any comprehensive/modern/detailed radiosity tutorials this is possibly the most pressing matter in the future works section. Techniques can flourish and evolve if people have easy access to resources about them. Unfortunately, even after a thorough search this seems to not be the case for radiosity. It should be remedied as soon as possible. The full code is planned to be open sourced as well.

7.5 Long-term plans

Integrate a hardware-accelerated ray tracer into the renderer: as described in the report, radiosity only handles diffuse reflections. In order to be able to handle specular reflections another global illumination technique is needed. Since ray tracing is suitable for the specular part it seems like a convenient choice. Depending on future decisions even more techniques could be implemented and the project could function as a "global illumination showcase" program.

Migrate to state of the art graphics APIs: Even though this project was written in modern OpenGL there are even newer APIs that might be worth investigating. In more concrete terms, DirectX 12 and Vulkan provide a lower level way to interface with GPUs and can provide significant performance benefits if used well. Additionally, there seem to be no radiosity implementations for either APIs so it is not known how well radiosity would perform on them. The downside is that these APIs greatly differ from OpenGL so porting the existing program is unlikely to be a small task.

Chapter 8

Conclusions

Global illumination algorithms are vast, complex algorithms that require a great deal of knowledge from multiple fields of science and nearly all of them have their own disadvantages. For calculating complex diffuse reflections, however, radiosity is still a possible candidate, even after more than 30 years since its initial introduction. Additionally, due to the evolution of computer hardware more and more complex scenes can be rendered. The GPU of a computer is capable of accelerating calculations greatly and also allows the programmer to easily define the number of desired patches within the scene by decoupling the geometry (triangles) from the radiosity elements (texels). The long history of radiosity does not come without a cost, however. Many conflicting research papers have been released on the subject and it can be hard to decide what is relevant for a modern implementation and what is not. Unfortunately, the OpenGL rasteriser is not suitable for the faster hemisphere-based radiosity implementation without producing an unacceptable amount of artifacts. The hemicube-based one, however, eliminates these artifacts without a huge performance hit. It is also proven that modern "core profile" OpenGL is suitable as a base for a radiosity renderer since the GPU pipeline can parallelise operations and can efficiently render and calculate light values for every patch within a given scene.

References

[1] Cindy M Goral, Kenneth E Torrance, Donald P Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *ACM SIGGRAPH Computer Graphics*, volume 18, number 3, pages 213–222. ACM, 1984.

[2] AUTODESK. Indirect (global) vs. direct illumination. `https://knowledge.autodesk.com/support/maya-lt/learn-explore/caas/CloudHelp/cloudhelp/2015/ENU/MayaLT/files/BoL-Indirect-global-vs-direct-illumination-htm.html`. Accessed: 2018-04-04.

[3] Henri Gouraud. Computer display of curved surfaces. Technical report, UTAH UNIV SALT LAKE CITY COMPUTER SCIENCE DIV, 1971.

[4] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[5] James F Blinn. Models of light reflection for computer synthesized pictures. In *ACM SIGGRAPH Computer Graphics*, volume 11, number 2, pages 192–198. ACM, 1977.

[6] Lance Williams. Casting curved shadows on curved surfaces. In *ACM Siggraph Computer Graphics*, volume 12, number 3, pages 270–274. ACM, 1978.

[7] Rouslan Dimitrov. Cascaded shadow maps. *Developer Documentation, NVIDIA Corp*, 2007.

[8] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, 1980.

[9] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, number 4, pages 143–150. ACM, 1986.

[10] Henrik Wann Jensen and Niels Jørgen Christensen. A practical guide to global illuminationusing photon maps. In *ACM Siggraph 2000, Course note 8*, 2000.

[11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, number 7, pages 1921–1930. Wiley Online Library, 2011.

[12] Sam Lapere. GPU path tracing tutorial 1: Drawing First Blood. `http://raytracey.blogspot.hu/2015/10/gpu-path-tracing-tutorial-1-drawing.html`. Accessed: 2018-04-02.

[13] Learn Computer Graphics From Scratch! `http://www.scratchapixel.com/`. Not just global illumination tutorials but does cover ray/path tracing extensively. Accessed: 2018-04-02.

[14] Károly Zsolnai-Fehér and Thomas Auzinger. Rendering course. Technische Universität Wien, `https://www.cg.tuwien.ac.at/courses/Rendering/VU.SS2018.html`. General rendering course but also heavily focuses on ray/path tracing. Accessed: 2018-04-02.

[15] NASA. A Lambertian Reflector. `https://landsat.gsfc.nasa.gov/a-lambertian-reflector/`. Author of source book: Dr. Nicholas M. Short. Author of online article: NASA. Accessed: 2018-04-02.

[16] Philip Dutré. Global illumination compendium. *Computer Graphics, Department of Computer Science, Katholieke Universiteit Leuven*, 2003.

[17] Professor Barbara Cutler. The Rendering Equation & Radiosity II. Rensselaer Polytechnic Institute `https://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S08/lectures/13_radiosity_II.pdf`. Accessed: 2018-04-02.

[18] McGuire Computer Graphics Archive, containing the Crytek Sponza scene. `https://web.archive.org/web/20180110180113/http://g3d.cs.williams.edu/g3d/data10/index.html`. Accessed: 2018-04-01.

[19] Unigine superposition benchmark. `https://benchmark.unigine.com/superposition`. The triangle count is not clearly stated on the website, but since the base version of the program is free it can be installed and checked within the renderer. Accessed: 2018-04-01.

[20] Michael F Cohen, Shenchang Eric Chen, John R Wallace, and Donald P Greenberg. A progressive refinement approach to fast radiosity image generation. *ACM SIGGRAPH computer graphics*, 22(4):75–84, 1988.

[21] Michael F Cohen and Donald P Greenberg. The hemi-cube: A radiosity solution for complex environments. In *ACM SIGGRAPH Computer Graphics*, volume 19, number 3, pages 31–40. ACM, 1985.

[22] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 49–56. ACM Press/Addison-Wesley Publishing Co., 1997.

[23] Michael Mara, Morgan McGuire, Derek Nowrouzezahrai, and David Luebke. Fast global illumination approximations on deep g-buffers. *NVIDIA Technical Report NVR-2014-001*, 2014.

[24] John R Wallace, Kells A Elmquist, and Eric A Haines. A ray tracing algorithm for progressive radiosity. In *ACM SIGGRAPH Computer Graphics*, volume 23, number 3, pages 315–324. ACM, 1989.

[25] Jarred Walton, Tuan Nguyen for www.pcgamer.com. Nvidia releases Volta-based Titan V for pc. `https://www.pcgamer.com/nvidia-releases-volta-based-titan-v-for-pc/`, 2017. Accessed: 2018-04-01.

[26] Matt Wuebling for Nvidia. More Extreme in Every Way: The New Titan Is Here – NVIDIA TITAN Xp. `https://blogs.nvidia.com/blog/2017/04/06/titan-xp/`, 2017. Accessed: 2018-04-01.

[27] Mark Tyson for www.hexus.net. Intel Core i7-8700K SiSoft Sandra benchmark results spotted. `http://hexus.net/tech/news/cpu/109487-intel-core-i7-8700k-sisoft-sandra-benchmark-results-spotted/`, 2017. Accessed: 2018-04-01.

[28] Rendering Pipeline Overview. `https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview`. Accessed: 2018-04-01.

[29] Robert Menzel. Understanding the parallelism of GPUs. `http://renderingpipeline.com/2012/11/understanding-the-parallelism-of-gpus/`, 2012. Accessed: 2018-04-01.

[30] Professor Colin Cooper. 6CCS3PAL Parallel Algorithms: Cluster Models lecture slides. King's College London, 2017.

[31] Joey De Vries. Debugging. https://learnopengl.com/In-Practice/Debugging. There is some more overlap between the debugging techniques here and in the report. However, only the framebuffer print technique was taken from the page. Accessed: 2018-04-01.

[32] Unwrapping UVs for Lightmaps. `https://docs.unrealengine.com/en-us/Engine/Content/Types/StaticMeshes/LightmapUnwrapping`. Accessed: 2018-04-02.

[33] Greg Coombe and Mark Harris. Global illumination using progressive refinement radiosity. *GPU Gems 2*, pages 635–647, 2005.

[34] Alex Wiltshire for www.pcgamer.com. How many frames per second can the human eye really see? `https://www.pcgamer.com/how-many-frames-per-second-can-the-human-eye-really-see/`. Accessed: 2018-04-04.

[35] László Szirmay-Kalos, László Szécsi, and Mateu Sbert. GPUGI: Global Illumination Effects on the GPU. In *Eurographics (Tutorials)*, pages 1201–1278, 2006.

[36] Dr Richard Overill. 6CCS3GRS Computer Graphics Systems: Colorimetry lecture slides. King's College London, 2017.

[37] Bill Licea-Kane. GLSL: Center or Centroid? (Or When Shaders Attack!). `https://www.opengl.org/pipeline/article/vol003_6/`, 2007. This is a relatively old article but the centroid sampling explanation is still valid. Accessed: 2018-04-04.

[38] Understanding gamma correction. `https://www.cambridgeincolour.com/tutorials/gamma-correction.htm`. Accessed: 2018-04-01.

[39] Cornell University. The cornell box. `http://www.graphics.cornell.edu/online/box/`. Accessed: 2018-04-02.

[40] Noel Llopis. When Is It OK Not To TDD? `http://gamesfromwithin.com/when-is-it-ok-not-to-tdd`, 2010. Accessed: 2018-04-02.

[41] Using UVAtlas (Direct3D 9). `https://msdn.microsoft.com/en-us/library/windows/desktop/bb206321(v=vs.85).aspx`. Accessed: 2018-04-01.

[42] Bernhard Korte and Jens Vygen. Bin-packing. In *Combinatorial Optimization*, pages 489–507. Springer, 2018.

Appendix A

Extra Material

A.1 Visibility calculation with a hemicube - tutorial

Note: this tutorial is meant to be the 7th or 8th in a series of tutorials thus might refer to concepts/code not presented here. In order to be more engaging for a more general graphics programmer audience its language is also a lot more informal than that of the rest of the report. The code formatting presented here is also slightly more rough due to the formatting constraints of the report. Sources are presented as a "Further reading" section, with links to the relevant material.

Welcome back everyone, last time we had a look at the geometric factor calculation for every receiving patch and made the first version of a working radiosity renderer! Congratulations for making it this far, the end is in sight already. However, as you might've noticed, the renderer produces no shadows, which is a *bit* of a problem. We're going to rectify that today. If you're somewhat familiar with radiosity papers you've most likely come across the terms "hemisphere" and "hemicube". There is plenty of material about how they are supposed to work *in theory* so if you really want to dig deep you can look them up in-depth. Here I'll cover how visibility calculation with a hemicube can be implemented. Personally I tried using the hemisphere method as well but the amount of artifacts produced were beyond acceptable, so with OpenGL using a hemicube makes more sense. For our current purposes it is enough to know that a hemicube is a cube cut in half. It can be used to check what is visible from a given patch and

receivers can be checked for visibility (with the normal Z-buffer shadow mapping algorithm). Without further ado, let's begin.

We could postpone this step but it'd only make it more painful so let's dive into vectors. In order to be able to build a robust hemicube solution we need to account for patches looking in *every* direction. To provide a bit more context: we basically want to create a virtual camera for every patch (i.e. something similar to what we observe the scene through). When cameras are created programmers usually define a "worldUp" vector that points to positive Y (0, 1, 0). This is needed so the orientation of the camera can be decided, since in 3D space there is an infinite amount of vectors that are at a right angle to the direction vector we want to look along and all of them could be the up vector. With this worldUp vector first a rightVector (which is at a right angle to the camera and as the name implies, to the right) can be calculated by taking the cross product of the direction vector and the worldUp vector. From this vector we can calculate the upVector of the camera by taking the cross product of the rightVector and the direction vector. Don't forget to normalise the vectors after each step. In code constructing a camera with the help of the glm::lookAt(eyePosition, cameraCenter, upVector) function would look like this:

```
1 glm::mat4 viewMatrix = glm::lookAt(cameraPosition, cameraPosition +
      ↪ directionVector, upVector);
```

Easy, right? This works for a camera but what happens if our patch has a direction vector that is parallel with our worldUp? If you perform the previously described steps with a vector like that (i.e. (0, 1, 0)) and our worldUp kept at (0, 1, 0) you are going to end up with (0, 0, 0) as our upVector. Which is not good, because the glm::lookAt() function's upVector cannot be (0, 0, 0). This problem is further exacerbated when we are calculating the vectors for the hemicube, since we need to juggle 5 vectors (one for each face of the hemicube) and an upVector for each of them. We have to approach this problem in a slightly more clever way.

What if we decomposed this problem into two stages? One for handling the worldUp vector and another one to handle the up vector for the hemicube (the hemicubeUp)? Enter the "double up" technique.

The first problem, the one with the worldUp can be fixed rather easily. We know our first solution fails in some cases, but how often does it fail? If the direction we want to look along (or indeed, *shoot* along, if we consider patches, which we do) is parallel with the defined worldUp.

If we normalise the direction vector it can be seen that this happens in exactly 2 cases. If the direction vector is (0, 1, 0) or (0, -1, 0). We can simply redefine our worldUp if one of those cases happens:

```
glm::vec3 worldUp = glm::vec3(0.0f, 1.0f, 0.0f);

//This is our normalised direction vector
glm::vec3 normalisedShooterNormal = glm::normalize(shooterWorldspaceNormal);

if (normalisedShooterNormal.x == 0.0f && normalisedShooterNormal.y == 1.0f &&
    ↪ normalisedShooterNormal.z == 0.0f) {
  worldUp = glm::vec3(0.0f, 0.0f, -1.0f);
}
else if (normalisedShooterNormal.x == 0.0f && normalisedShooterNormal.y ==
    ↪ -1.0f && normalisedShooterNormal.z == 0.0f) {
  worldUp = glm::vec3(0.0f, 0.0f, 1.0f);
}
```

Using this "sanitised" worldUp we can define a rightVector and an upVector for the hemicube:

```
glm::vec3 hemicubeRight = glm::normalize(glm::cross(shooterWorldspaceNormal,
    ↪ worldUp));
glm::vec3 hemicubeUp = glm::normalize(glm::cross(hemicubeRight,
    ↪ shooterWorldspaceNormal));
```

From this point only a few basic operations remain. For the front, left and right faces we can use the hemicubeUp as the upVector. For the direction vectors, the front face can use the normal vector of the shooter, the left face can use the negative hemicubeRight and the right face can use the hemicubeRight. This is what it looks like in code:

```
glm::mat4 frontShooterView = glm::lookAt(shooterWorldspacePos,
    ↪ shooterWorldspacePos + shooterWorldspaceNormal, hemicubeUp);

glm::mat4 leftShooterView = glm::lookAt(shooterWorldspacePos,
    ↪ shooterWorldspacePos + (-hemicubeRight), hemicubeUp);
glm::mat4 rightShooterView = glm::lookAt(shooterWorldspacePos,
    ↪ shooterWorldspacePos + hemicubeRight, hemicubeUp);
```

Only the up and down faces remain. For these the direction vector can be the hemicubeUp and negative hemicubeUp, respectively. The up vectors can be the negative normalised direction vector and the normalised direction vector, respectively.

```
1 glm::mat4 upShooterView = glm::lookAt(shooterWorldspacePos,
      ↪ shooterWorldspacePos + hemicubeUp, -normalisedShooterNormal);
2 glm::mat4 downShooterView = glm::lookAt(shooterWorldspacePos,
      ↪ shooterWorldspacePos + (-hemicubeUp), normalisedShooterNormal);
```

And we're done with the hard part! Next take a look at the function declaration:

```
1 std::vector<unsigned int> createHemicubeTextures(ObjectModel& model,
      ↪ ShaderLoader& hemicubeShader, glm::mat4& mainObjectModelMatrix, std::
      ↪ vector<glm::mat4>& viewMatrices, unsigned int& resolution, glm::vec3&
      ↪ shooterWorldspacePos, glm::vec3& shooterWorldspaceNormal);
```

It is a handful but the only two notable things are: we return an std::vector of the unsigned ints that refer to the depth textures and since we use the view matrices we construct here in the updateLightmaps() function, we write to a glm::mat4& that contains the matrices.

Let's render the front face first, to see that we're heading along the correct path. We're using a 90° projection matrix to fit together the faces of the cube perfectly. Here I decided against using a cube map so there is going to be some code repetition. This is just one solution that is possibly far from the best so feel free to use a cube map or improve it in any way if you want to.

```
1  unsigned int hemicubeFramebuffer;
2  glGenFramebuffers(1, &hemicubeFramebuffer);
3
4  unsigned int frontDepthMap;
5  glGenTextures(1, &frontDepthMap);
6  glBindTexture(GL_TEXTURE_2D, frontDepthMap);
7  glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, resolution, resolution, 0,
       ↪ GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
8  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
9  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
10
11 glBindFramebuffer(GL_FRAMEBUFFER, hemicubeFramebuffer);
```

```
12 glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D,
       ↪ frontDepthMap, 0);
13 glDrawBuffer(GL_NONE);
14 glReadBuffer(GL_NONE);
15
16 glViewport(0, 0, resolution, resolution);
17
18 glClear(GL_DEPTH_BUFFER_BIT);
19
20 glm::mat4 shooterProj = glm::perspective(glm::radians(90.0f), 1.0f, 0.1f,
       ↪ 100.0f);
21
22 int lampCounter = 0;
23
24 for (unsigned int i = 0; i < model.meshes.size(); ++i) {
25
26   hemicubeShader.useProgram();
27
28
29   hemicubeShader.setUniformMat4("projection", shooterProj);
30
31   hemicubeShader.setUniformMat4("view", frontShooterView);
32
33
34   if (model.meshes[i].isLamp) {
35     glm::mat4 lampModel = glm::mat4();
36
37     lampModel = glm::translate(lampModel, lightLocations[lampCounter]);
38     lampModel = glm::scale(lampModel, glm::vec3(lampScale));
39
40     hemicubeShader.setUniformMat4("model", lampModel);
41
42     model.meshes[i].draw(hemicubeShader);
43
44     ++lampCounter;
45   }
46   else {
```

```
47      hemicubeShader.setUniformMat4("model", mainObjectModelMatrix);
48
49      model.meshes[i].draw(hemicubeShader);
50    }
51  }
```

For the vertex program we simply transform the vertices to the clip-space of the patch (i.e. what we could call light-space):

```
1  #version 450 core
2
3  layout (location = 0) in vec3 vertexPos;
4
5  uniform mat4 model;
6  uniform mat4 view;
7  uniform mat4 projection;
8
9  void main() {
10     gl_Position = projection * view * model * vec4(vertexPos, 1.0);
11 }
```

The fragment shader is a simple passthrough shader:

```
1  #version 450 core
2
3  void main() {
4
5  }
```

In our lightmapUpdate fragment shader we define a function that performs the shadow mapping procedure. It returns a float, since we use percentage-closer filtering (PCF) with 25 samples in order to smooth out the shadow edges:

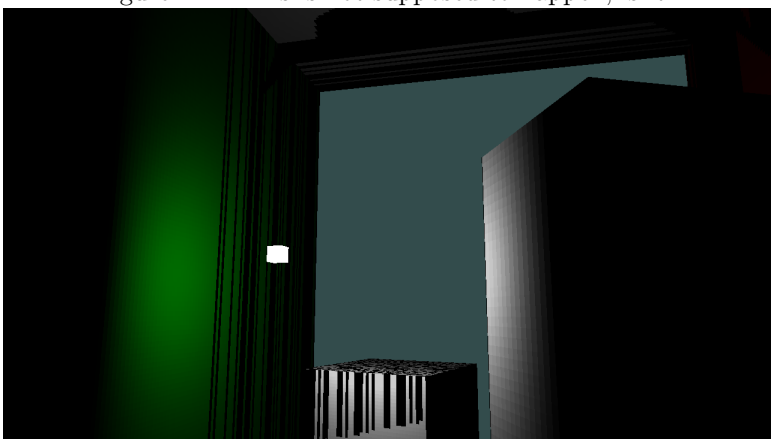
```
1  //Parts of this function adapted from https://learnopengl.com/Advanced-
       ↪ Lighting/Shadows/Shadow-Mappings
2  float isVisible(sampler2D hemicubeFaceVisibilityTexture, vec4
       ↪ hemicubeFaceSpaceFragPos) {
3
```

```
    //Perform the perspective divide and scale to texture coordinate range
    vec3 projectedCoordinates = hemicubeFaceSpaceFragPos.xyz /
        ↪ hemicubeFaceSpaceFragPos.w;
    projectedCoordinates = projectedCoordinates * 0.5 + 0.5;

    float closestDepth = texture(hemicubeFaceVisibilityTexture,
        ↪ projectedCoordinates.xy).r;
    float currentDepth = projectedCoordinates.z;

    float shadow = 0.0;

    //Use percentage-closer filtering (PCF), 25 samples
    vec2 texelSize = 1.0 / textureSize(hemicubeFaceVisibilityTexture, 0);
    for(int x = -2; x <= 2; ++x) {
        for(int y = -2; y <= 2; ++y) {
            float pcfDepth = texture(hemicubeFaceVisibilityTexture,
                ↪ projectedCoordinates.xy + vec2(x, y) * texelSize).r;
            shadow += (currentDepth) > pcfDepth ? 0.0 : 1.0;
        }
    }

    shadow = shadow / 25.0;

    return shadow;
}
```

Call this function in main() and multiply the form factor value we got in the previous tutorial by the visibility value. Don't forget to bind the depth texture, set the shooterProjection matrix and the front view matrix in updateLightmaps()! Modify the vertex and fragment shaders accordingly, including transforming the input vertices to the light-space of the hemicube.

If you did everything correctly run your program and congratulations, we're done with most of the wor...oh wait a minute...

Figure A.1: This is not supposed to happen, is it?



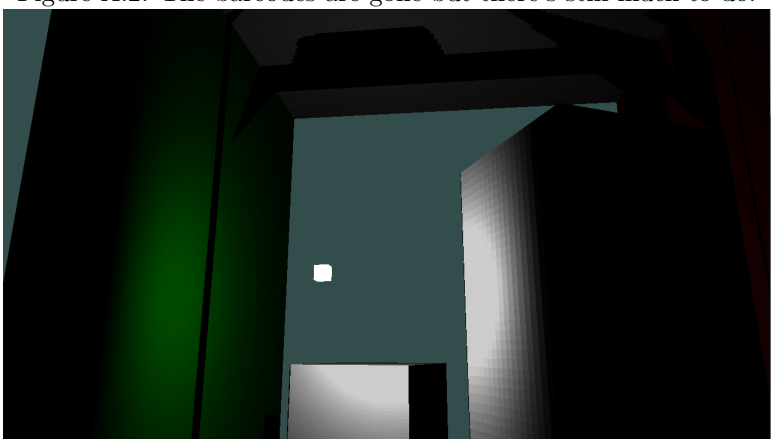
The answer is, no, it isn't. Before you go looking through every single line of code multiple times, trying to find the bug (like I did when I encountered this scene for the first time) I'll go ahead and spoil it that there's nothing wrong with our code. Well, there is, but it is due to a lack of a few lines of code rather than us having made a mistake in the existing codebase. Let's go and find out what we're missing, shall we?

First, what you see above is a nasty mixture of 3 bugs. In no particular order, let's fix them. Leading the pack is going to be the barcode-like artifacts all over the scene. Believe it or not, those bars are not more than a really common shadow mapping error, shadow acne. Because of our use of lightmaps instead of the normal acne-like appearance it took on a form of black bars. The fix is now easy, we need to revisit the isVisible() function to add a bit of bias and modify how we check the depth map:

```
1  float bias = 0.001;
2
3  ...
4
5  vec2 texelSize = 1.0 / textureSize(hemicubeFaceVisibilityTexture, 0);
6  for(int x = -2; x <= 2; ++x) {
7    for(int y = -2; y <= 2; ++y) {
8      float pcfDepth = texture(hemicubeFaceVisibilityTexture,
           ↪ projectedCoordinates.xy + vec2(x, y) * texelSize).r;
9      shadow += (currentDepth - bias) > pcfDepth ? 0.0 : 1.0;
10   }
11 }
```

The bias here is nothing complicated but works generally well for the Cornell-box-like test scene. If you want to be more fancy with it (i.e. modify it depending on the angle between the surface and the light emitter), feel free to!

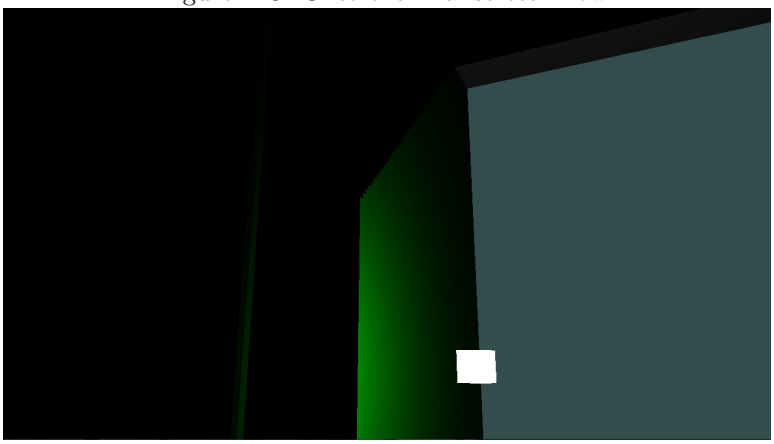
Figure A.2: The barcodes are gone but there's still much to do.



The second issue is called oversampling. When we sample the depth map some fragments are sampled outside the range of the depth map. The depth values retrieved in cases like this by default is maximum depth, which means that these fragments are always going to be marked visible. Again, the fix is not hard: specify a border "colour" (in this case more like border depth) of minimum depth then clamp the depth map to this border. With this fix the depth map is declared and initialised this way:

```
1  float borderColour[] = { 0.0f, 0.0f, 0.0f, 1.0f };
2
3  unsigned int frontDepthMap;
4  glGenTextures(1, &frontDepthMap);
5  glBindTexture(GL_TEXTURE_2D, frontDepthMap);
6  glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, resolution, resolution, 0,
       ↪ GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
7  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
8  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
9
10 //This part is needed to avoid light bleeding by oversampling (so sampling
       ↪ outside the depth texture)
11 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
12 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
13 glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, borderColour);
```

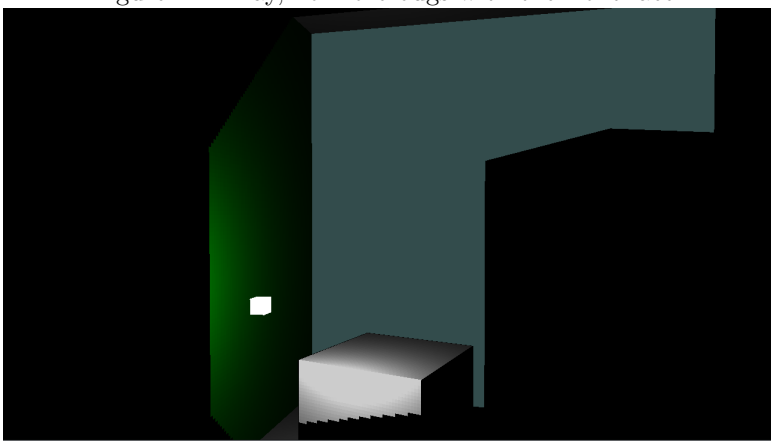
Figure A.3: Onto the final stretch now.



There is still an annoying bar of light where it should not be, directly in the plane of the light source. It is caused by the fragments that are in the aforementioned plane having a depth value of 0 compared to the light emitter (since they are in the same plane). Once again, let's revisit the isVisible() function and append a small section to it, that adds our bias to the fragment and checks if it maps to the plane of the shooter (i.e. 0.0) or behind that point:

```
1  ...
2  shadow = shadow / 25.0;
3
4  float zeroDepth = currentDepth - bias;
5
6  //If the current fragment is in one plane or behind the shooter, it is in
       ↪ shadow
7  if (zeroDepth <= 0) {
8      //Zero means it is in shadow
9      shadow = 0.0;
10 }
11
12 return shadow;
```

Figure A.4: Yay, no more bugs with the front face!



A small note: my renderer didn't have PCF implemented when these screenshots were taken so do excuse the harder shadow edges.

Now do the same procedure with the other four faces and you're done for real now. You implemented what is most likely the hardest functionality presented in this tutorial series. Well done! Next time we'll put the whole algorithm together and get the iteration working. Afterwards we'll look at ways we can improve the renderer, which is going to involve an encounter with texture-space multisampling and custom multisampled texture resolve operations. Until then, stay tuned and thank you very much for reading!

Further reading

If you want to know more about the theory behind this tutorial here's the original paper about the radiosity hemicube: `http://artis.imag.fr/~Cyril.Soler/DEA/IlluminationGlobale/Papers/p31-cohen.pdf`

If you need a reminder about how traditional shadow mapping works take a look here: `https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping`

A.2 Links to other renderers

Here are a few radiosity renderers, mostly ones that had screenshots available that showed a test scene similar to the Cornell box used for this project.

http://www.cs.unc.edu/~jpool/COMP870/Assignment3/ As far as can be told this renderer uses the full-matrix classical radiosity.

https://docs.blender.org/api/htmlI/x8846.html Radiosity in Blender. Unfortunately, recent Blender versions seem to lack the radiosity feature.

http://www.captaindeath.com/projects/detail.aspx?proj_id=2 Radiosity for a game engine, from 2006.

http://www.cs.unc.edu/techreports/03-020.pdf Both of the researchers who wrote GPU Gems 2 Chapter 39 had also worked on this radiosity renderer.

To conclude, links to the two renderers this implementation is inspired by, even if they provide no screenshots of scenes that would closely match the classical Cornell box:

https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter39.html GPU Gems 2 Chapter 39. It partly formed the theoretical basis of the 1st implementation and the advanced shooter selection.

http://sirkan.iit.bme.hu/~szirmay/gpugi1.pdf GPUGI. The patchless radiosity idea forms the basis of the patch representation within the renderer developed for this project. However, there are notable differences elsewhere, for example in the shooter selection. A Cornell box-like scene can be seen in Figure 47.

Appendix B

User Guide

B.1 System requirements

The development machine was a laptop with the following specifications:

- OS: Windows 7 Ultimate, 64 bit
- RAM: 12GB DDR3
- CPU: Intel Core i7 2630QM
- GPU: Nvidia GeForce GTX 560M - driver version: 372.70

Since the project only had access to the aforementioned PC these system requirements might not be comprehensive or completely accurate but in general at least a PC with the following is needed:

- OS: Windows 7 64 bit or newer 64 bit Windows versions.
- RAM: DDR3 or DDR4, 4 to 8 GB at least, the more the better.
- CPU: Intel or AMD desktop/laptop chip but no further restrictions, the faster the better.
- GPU: A GPU that supports OpenGL 4.5 is needed. That is, at least an Nvidia GeForce GT 420 or AMD equivalent. A fast GPU should make the program run significantly better. Make sure to use up-to-date drivers since old drivers might not support OpenGL 4.5, even if the card supports it.

- Microsoft Visual C++ 2017 redistributable has to be installed.

B.2 Usage instructions

The user has to select a scene file to be loaded (which has to be a .obj file with a corresponding .mtl file) by clicking on the corresponding button on the user interface, then following the dialogue prompt. This scene has to have textures for every mesh and complete, per-mesh non-overlapping UV mapping. If texture filtering and multisampling are desired to be used then gaps should be created between the UV maps of individual faces (in this case a face can be a side of a cube, for example).

In the settings menu the user can:

- Set the resolution of the renderer.
- Change the resolution of the lightmaps: higher means better quality lighting but much slower radiosity calculations.
- Change the light attenuation type:
  - Linear- light gets weaker linearly with distance. Weak attenuation, leads to bright scenes.
  - Quadratic - light gets weaker with distance squared. Medium attenuation, leads to medium-bright scenes.
  - Quadratic with patch size correction - quadratic * π attenuation, multiplied by the size of the individual patches. Overall it leads to strong attenuation and relatively dark scenes.
- Continuously update lightmaps: if ticked the scene is updated as the radiosity calculation goes along. If unticked the scene is updated after every patch has shot its energy on the shooter mesh. The latter is slightly faster.
- Texture filter lightmaps: makes lightmaps smoother but depending on the UV mapping it might cause artifacts. See previous paragraph.
- Use multisampling: reduces the amount of lightmap artifacts at UV edges. Depending on the UV mapping it might introduce different artifacts. See previous paragraph.

The settings window can be closed when done and the renderer started. Within the renderer the user can navigate the scene with the "**WASD**" keys and the mouse. The user can place a light source at the current location of the camera by pressing the left mouse button. Before the radiosity iteration but after the light placement the user needs to initiate a preprocess function by pressing "**E**". After this the radiosity iteration is started with the right mouse button. The whole list of the key bindings is:

- WASD and mouse: navigation.
- ESC: quit program.
- U: switch on ambient light.
- I: switch off ambient light.
- Left mouse button: place light source at current location.
- E: initiate preprocess step. Can only be done once.
- Right mouse button: start radiosity iteration.
- Q: stop radiosity iteration.
- R: reload every currently loaded shader program.

Troubleshooting:

If the program complains about a missing dll (zlib.dll) either compile this dll from source (the 64 bit version) or download the compiled version from here: `https://francescofoti.com/2013/12/compiling-the-zlib-compression-library-with-visual-studio-2013/`. Rename the 64 bit version file to zlib.dll (or to the name the program wrote which can be a variation of zlib.dll, e.g. zlib1.dll) and copy this file into the folder of the executable.

B.3 Compilation instructions

The project uses the following libraries:

- GLFW 3.2.1: `http://www.glfw.org/download.html`

- GLEW 2.1.0: `http://glew.sourceforge.net/`
- wxWidgets 3.10: `https://www.wxwidgets.org/`
- Assimp 4.0.1 `http://assimp.org/index.html`
- GLM 0.9.8.5 `https://glm.g-truc.net/0.9.8/index.html`
- stb_image.h `https://github.com/nothings/stb/blob/master/stb_image.h`

It is recommended to use Microsoft Visual Studio 2017, with C++ 17 support installed and use the included VS Solution.

The last two libraries are header-only libraries and thus do not need to be built. The others are recommended to be built from the source code with 64 bit debug builds. Place the built libraries into externalDependencies\libs\DebugLibs. Place the Assimp dll into the folder where the executable of the program is located (by default OpenGLRadiosityRenderer\bin). Place the headers from all of the libraries with their directories into externalDependencies\include, aside from stb_image.h, which should just be placed into the aforementioned folder. Open the VS solution, make sure the linker and the include directories are set up properly and that the project uses C++ 17. If everything is set up correctly the project should successfully compile/build. Use the 64 bit debug build configuration.