UNIVERSITY OF OSLO
Department of Informatics

Variably sized filter kernels for GPU Accelerated Photon Mapping

Erik W. Bjønnes
erikwb@ifi.uio.no

December 15, 2010

Abstract

This thesis focuses on how it is possible to find a variable filter kernel size, without the use of a *k*d-tree for use when accelerating photon mapping on the GPU. First we take a look at two different ways of making use of the GPU when implementing photon mapping before we present two new solutions to how this can be done using only data structures native to the GPU.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To generate a believable scene with computer graphics, it is important that the light looks believable. The most noticeable part of this is having correct shadows. If the shadows are not present, or are off, it will be quite easy to spot that this is not a real scene. Indirect light is another important part that makes a scene believable. With indirect light we mean light that has hit a surface and bounced from one to many different surfaces, either by reflection or refraction, with the results depending on the material characteristics of the surfaces which has been hit on the way.

Imagine a completely dark room, with a flashlight pointing at one wall. Turning the flashlight on, we can see that even though only one wall is in light, the rest of the room is, to some extent, lit up making it possible to spot other objects in the room. Now imagine the flashlight shining through a body of water. When the flashlight is turned on a pattern of light appears on the other side of the water. And when waves pass across the surface of the water, the pattern chances. This, caustics, is another form of indirect light, where the light is refracted and reflected by both the glass and the water focusing the light on some parts more than others. See Figure 1.1 for an example of photon mapping with indirect light and caustics.

Whilst such effects might not be readily noticeable in a scene, though the absence of them will make it much easier to spot that the scene is not real. There are several different ways of calculating indirect light, here we will focus on one algorithm called photon mapping. Photon mapping, as the name implies, calculates the indirect light by tracing photons through the scene. This is done by emitting photons from each light in the scene, tracing their path through reflections and refractions through the scene, storing the position where the photon is absorbed. The indirect light of a given position is then found by sampling the nearby photons and calculating their contribution. To reduce visual artifacts, and give a smooth look to the scene, each pixel in the scene is affected by a predetermined number of photons. This is done by sampling photons outward from the pixels position until enough photons have been found, and then the radius needed to find these photons as the radius for the blur filter, which blurs the photons into the scene. This gives a small radius for areas where the photon density is high, and a larger area for areas with sparse photon coverage.

Photon mapping originated as a CPU technique, though the advent of programmable graphics hardware have allowed for alternative implementations making use of the GPU to accelerate the algorithm. This allows for a potentially large increase in speed since each photon can be traced in parallel, though not without a cost. Due to the architectural differences between the

Figure 1.1: A picture of a scene using photon mapping for indirect light and caustics. Image courtesy of Dyken [3]

CPU and GPU, the GPU has a different set of constraints on code complexity and memory access, it is not simply a matter of recompiling code for the GPU and it will be able to run much faster. Techniques have to be tailored to make use of the GPUs parallel nature by aligning memory accesses and using independent kernels, and then it is possible to get drastic speed increases compared to a CPU version.g

In this thesis we will show two new techniques for finding a variable filter size when implementing photon mapping on the GPU. They are primarily aimed at solutions built around splatting, though it should be possible to extended them for use with other solutions.

To get the most from this thesis it is recommended with a basic background in Computer Graphics. In the background chapter we do give an introduction to the topics needed to follow the algorithms, and most concepts have references to the articles introducing the topic, though most of them require prior knowledge.

1.1 Related work

"Photon Mapping on Programmable Graphics Hardware" by Purcell et. al. [14] is an early example on what has been termed GPGPU, an acronym of General-purpose computing on graphics processing units. This means they use the vertex and fragment shaders to calculate the photons path through the scene. In short their approach is to try fitting the CPU algorithm to the GPU, trying out different data structures in which to store the photons, as well as two different approaches for an varying blur filter kernel radius. This article is a good example of how alternative data structures are needed when moving a technique from the CPU to the GPU. As an early example of GPGPU programming, it is under even stricter constraints for data structures than the case is with the hardware of today and corresponding programming models. However, it does show how graphics based data structures and fixed function hardware can be used in alternate ways, which is still applicable today. We will go through the paper in some detail in Section 2.5.

"Hardware-Accelerated Global Illumination by Image Space Photon Mapping" by McGuire and Luebke [9] show another way of accelerating photon mapping by using the GPU. Through analysing the algorithm they find the parts most suitable for implementation on the GPU, and keep the remaining part of the algorithm on the CPU. This hybrid approach makes use of the hardware rasteriser on the GPU to calculate the first bounce of the photons into the scene, before transferring the results to back to the CPU. There the approach is to keep tracing the photons as usual, before uploading the photons back onto to the GPU for the rasteriser to scatter them into the scene, a technique commonly known as "splatting".

Whilst making good use of the systems hardware, playing on the GPUs and CPUs strengths, it is still not making use of the full potential of either. By downloading the photons from the GPU to the CPU and then uploading them again to the GPU, it is constrained by the speed of the PCI-Express bus. And by scattering the photons with the rasteriser it forgoes the variable filter radius easily found when sampling the photons from the *k*d-tree on the CPU. We will cover the article in Section 2.6 and then show in Chapter 3 two solutions that can be used to extended it to use a variable filter width.

We are basing one of our solutions on using a Summed Area Table (SAT) as an intermediate

data structures. We have found two mentions of what appears to be a similar approach. The first is by Pham [11] who mentions that he is using SATs to count neighbours, without giving any further details what this is then used for. Herzog et. al. [7] is making use of multiple, what they call local, SATs for use when doing the radiance estimation, however it is unclear if they are using these to find varying filter sizes or using them to directly calculate contribution.

1.2 Scope

In this thesis we will show how a GPU implementation of photon mapping based on splatting, similar to McGuire and Luebkes [9] solution, can be extended to make use of a varying sized filter kernel without the use of a *k*d-tree. This should improve the implementation compared to making use of a fixed size filter kernel that needs to be tweaked for each scene. In addition to a naive brute force solution we have developed two new solutions, one utilising a density map with pyramidal mipmaps and one using a Summed Area Table (SAT) based on the density map. All three are designed to run on the GPU, making use of its parallel nature.

The naive brute force solution is just that, and is used as a baseline for comparisons with the other two. The pyramidal mipmap solution, built with a summation filter, is an attempt to make use of a GPU native data structure to speed up sampling. However, matching the sample area with what mipmap level to sample from and splitting the remaining sample area are complex operations and very taxing for the fragment shader, so performance can suffer. With the third solution, we are building our solution around a technique often used in computer imaging, SATs. Whilst it is costly to build the SAT, sampling a rectangular area is of constant complexity, making this the most suited solution of the three.

We will not cover how the GPU can be used to trace the photons path, as that topic is large enough to warrant a thesis on its own. For information on how this can be done we recommend looking a NVIDIAs GPU ray tracing library **Optix**[10]. We will simply limit ourselves to handling the photons once they have been traced through the scene.

To avoid introducing extra constraints and requirements to an existing render pipeline because of our solution, we limit ourselves to using the OpenGL graphics context for our solutions. This should help limit obstacles when integrating our solutions into existing graphics pipelines. The solutions should be easy to transfer to DirectX or a GPGPU context if needed. As a bonus, this self-imposed constraint should allow for a high occupancy of the GPU, since the 4 positions being calculated by the shading quad will in most cases have similar values and should therefore not diverge much from each other. It also means we do not have to find the ideal thread dimensions, as we only need to render a full screen quad with a size corresponding to the values we want calculated.

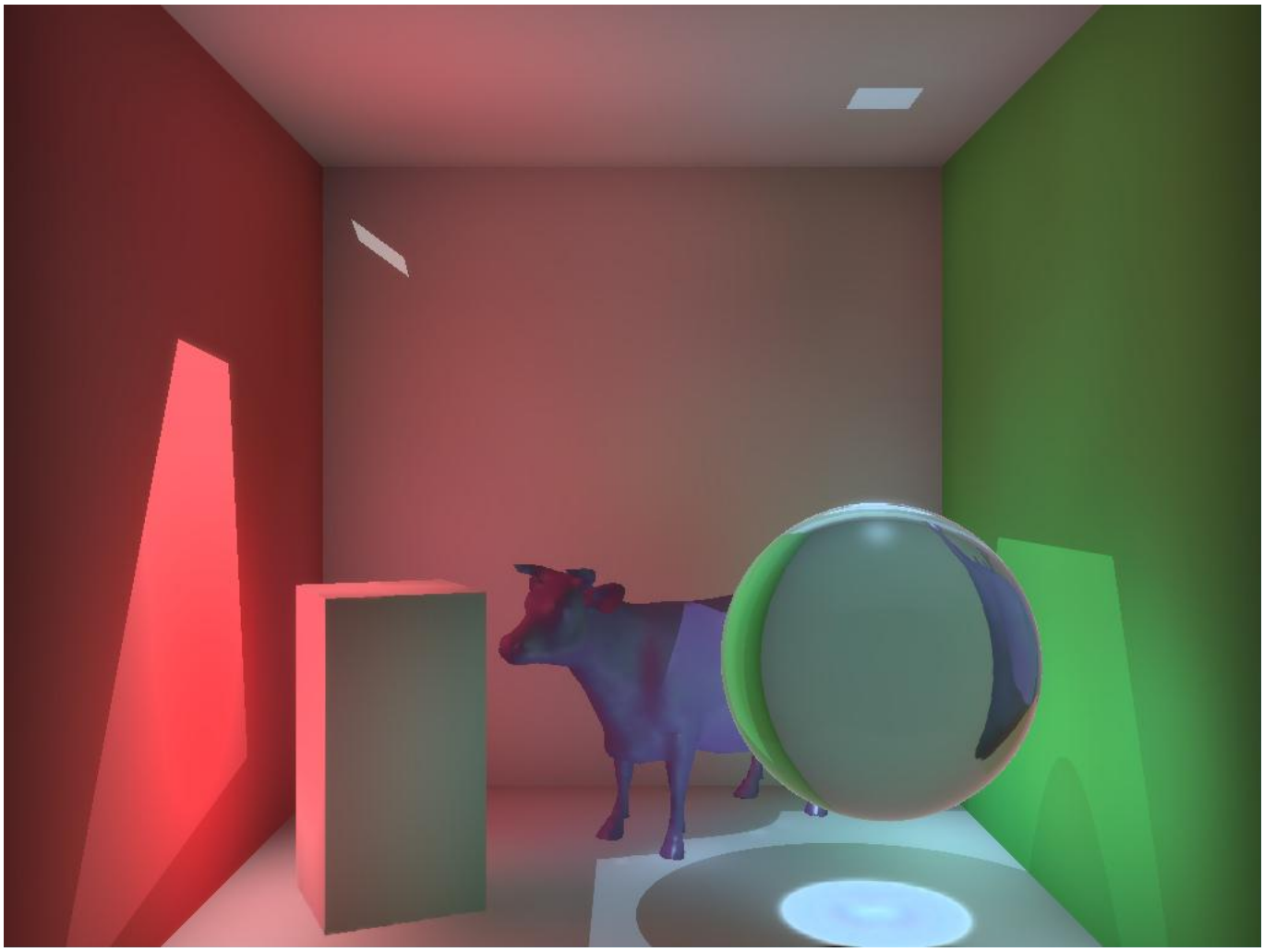


Figure 1.2: Image showing how a Cornell box can look when using a well chosen fixed size radius. Image from Luebke [8]

1.3 Problem description

We are interested in solving the following task: Given a set of screen space photons and a screen space position p, find a radius r that encompasses at least n photons. It should be done in a GPU-friendly way fitting into an existing graphics pipeline. To this end we will explore three different approaches to find these radii, both with and without using intermediate data structures.

When implementing photon mapping on a CPU, it has access to flexible data structures that allows it to easily find the blur filter radius for a given position by simply traversing the *k*d-tree until enough photons have been found.

Due to the different memory model on the GPU it is not efficient to make use of *k*d-trees on current hardware. It can be done, but it will not exhibit the same speed characteristics that we know from the CPU. So instead other data structures must be employed.

It is possible to skip this step altogether and instead make use of a fixed radius tweaked on a scene by scene basis, which is the approach taken by McGuire and Luebke [9], but this will

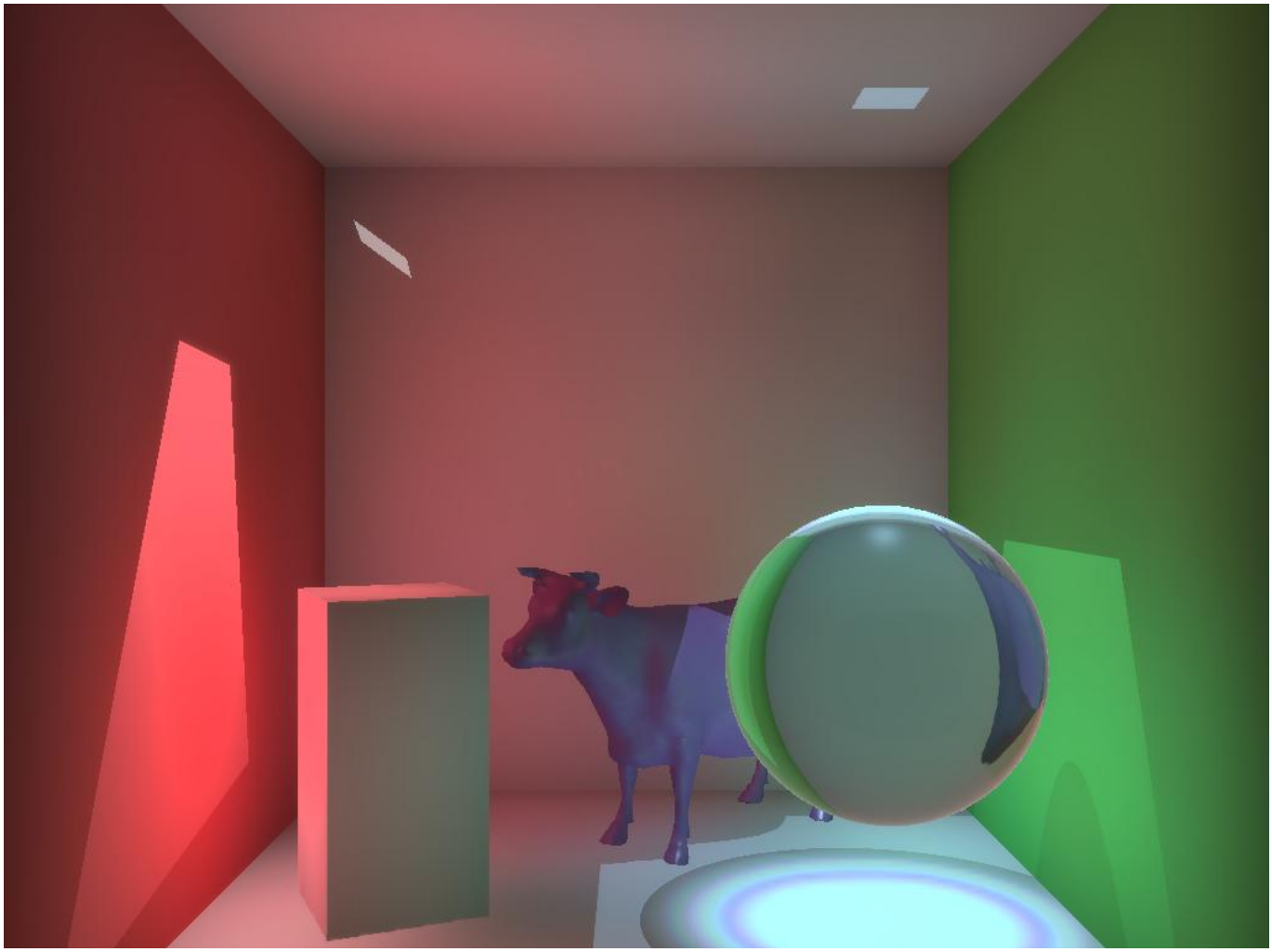


Figure 1.3: Image showing how details are lost when using a too large fixed radius, compared with Figure 1.3

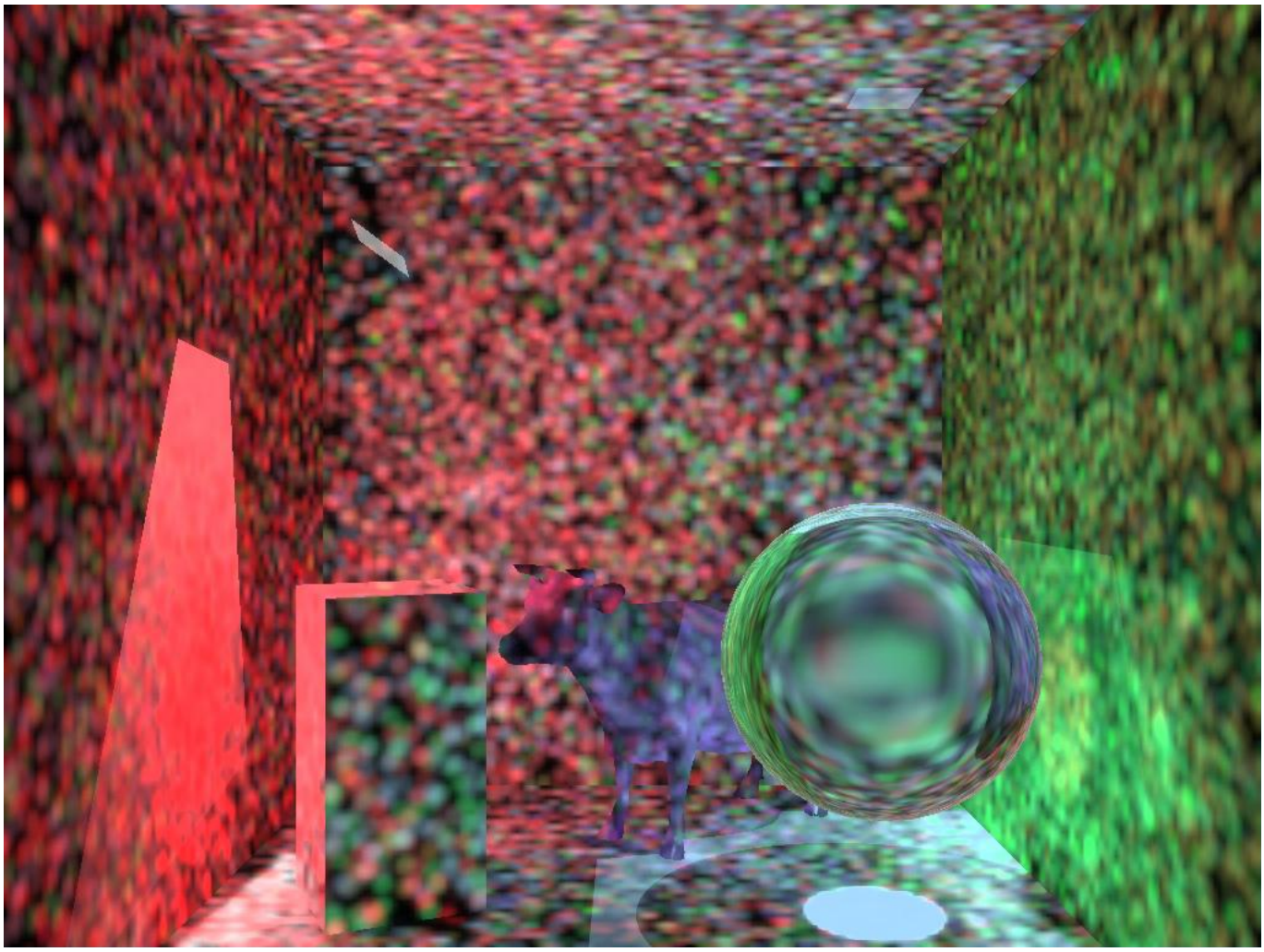


Figure 1.4: Image of scene using too small a fixed size filter radius. Taken from Luebke [8]

be a trade off between different types of noise in the resulting image. Use a too large radius and details will be lost in areas with dense photon concentrations. An example of this can be seen when comparing Figure 1.3 to Figure 1.3 where the details of the caustics are lost.

If the radius is too small, as seen in Figure 1.3, there will be areas not affected by enough photon. Depending on the complexity of the scene it can be impossible to find a suitable fixed width radius.

Chapter 2

Background

Here we will briefly cover topics relevant for both the thesis and algorithms we will be using, as well as two articles on using the GPU to accelerate photon mapping.

2.1 Textures

When implementing algorithms on GPUs we talk different data structures like maps, tables, textures and buffers, but in essence they are implemented all the same, a buffer storing 1...4 values of type t per position in memory. The name they are given is commonly used to reflect either content or usage, and we will now give a quick overview of the data structures relevant for this thesis.

The **density map** is simply a texture storing the number of photons at each position. It is created by rendering the photons final position, after they have been traced through the scene, and then simply storing 1 at the position. All these 1s are then added together, and we get the density map. Similarly the **radius map** is the smallest radius at screen space position p containing at least m photons.

The **photon map** differs from many other maps, in that it only specifies what is stored, the position, energy and direction of the incoming photon, and not how it is stored. For CPU implementations it is common to store the photon maps as kd-trees, splitting along the xyz axis. On the GPU some sort of ordered grid is often used.

A pyramidal **mipmapped texture** is a texture containing one or more levels, where each level above the base level, 0, is a quarter of the previous level commonly. Often thought of as a pyramid texture, where the position at one level contains the filtered sum of the 4 values from the level underneath. What filter is not specified, though it is commonly a $2x2$ box filter reduction. The size of a mipmapped texture is $2n$ where n is the size of the original texture. Each level in the mipmap pyramid has to have the same internal format as the base level, so if the base has a format for a single 32-bit floating point value, so does all the other levels. Though one can implement manual packing of values if needed. For more information see Section 6.2.2 in Akenine-Möller et.al. [1].

A **Summed Area Table (SAT)** is another type of mipmapped texture where the value at position p is the sum of all the previous values below and to the left of p. Introduced by Crow [2] as an alternative to pyramidal mipmaps, by allowing for more filtering options. It

enables one to sample an arbitrarily large area, within the bounds of the table, by taking only four samples. We cover it in some more detail, with figures, in Section 3.4.

2.2 Shadow Mapping

One way of adding shadows is to use **Shadow Mapping**. This technique creates a map of what objects are visible from a given light, with which comparisons can be done to see if a fragment is affected by the light or not. There are several variations of shadow maps, and we will only give a brief overview. For a more detailed cover we recommend Section 9.1.4 from Akenine-Möller et.al [1].

To generate a basic shadow map, a camera is placed at the lights position, looking along the direction of the light. The scene is then rasterised, with depth checking enabled, storing the final Z-depth of each fragment in the shadow map. Now, when rendering the full scene, the position of the fragment being shaded is transformed into light space and compared with the depth stored in the shadow map. If the depth of the position is smaller than the stored depth, it is in light, and is shaded as such. Should it be larger than the stored depth, it is in shadow and we ignore the light contribution. This is very similar to the initial step when tracing photons into the scene and can, like in McGuire and Luebke [9] solution, be combined.

One limitation of shadow mapping is that the quality of the shadows are dependent on the resolution of the shadow map, with a low resolution introducing aliasing artifacts. A higher resolution reduces the jagged edges of the shadows, at the cost of more space used in the GPUs memory as well as rendering speed. The resolution needed is also dependent on where the camera position is, due to the non-linear nature of Z-depth. Another limitation is that by storing the Z-depth, it is only well suited for lights pointing in one direction, like spot lights or directional lights. For point lights a cube map is needed.

One of the biggest problems with shadow mapping is self-shadow aliasing, commonly called "surface acne", where an object that should be in light has areas of shadow. There are several causes, such as lack of precision when doing depth comparisons, and sample positions not quite corresponding between rasterising from light position and camera position. The former is commonly resolved by adding a small offset to the depth, pulling the sampled depth forwards when doing the comparisons. The bias factor can either be a constant, or *depth slope scale bias* where it is dependent on the angle of the light to the object. See Akenine-Möller [1] page 351 for more information.

2.3 Rendering models

There are many different rendering models in use today aimed at solving different problems. The most commonly used raster based models are the traditional forward renderer and the deferred renderer, and we will cover them briefly in the following sections. We will also give a very brief overview of ray tracing.

Rasterisation

To transform a 3D scene into a 2D image there are different approaches available. A popular one in both computer games and computer generated movies is rasterisation. Simplified, the basic idea behind this technique is to transform the 3D scene from world space and into the 2D plane of screen space, storing only the fragments closest to the screen based on their position on the z-axis. Any previously stored fragment is simply discarded if another closer fragment is found. This is a relatively simple operation that can be done as a matrix vector multiplication.

Shaders are the programmable part of the GPU and are split into at least vertex shaders and fragment, or pixel, shaders. As GPUs have grown the number of programmable shading stages have grown, we also have geometry shaders in OpenGL 3.2 with two additional stages, tessellation control shaders and tessellation evaluation shaders in OpenGL 4.0 The solutions we present for finding the radius for a variable sized filter kernel only make use of vertex and fragment shaders, and the vertex shader is simply used as a pass through.

A **pixel-** or **shading quad** is the smallest granularity on the GPU. This means that if any fragment diverges from the others in the quad, the quad will perform suboptimal. Should one fragment take require more iterations than the others, the others will wait until all have finished before terminating, or continuing. For a more detailed explanation, see Akenine-Möller [1], page 867.

2.3.1 Forward rendering

Traditionally, the most common way of calculating real-time light contribution is forward rendering, where all of the geometry are rendered once for each light in the scene, which has the obvious drawback of the geometry being rendered multiple times. As each object is rasterised the resultant fragments are shaded, and then later on discarded if another fragment is closer to the screen, which means it is prone to overdrawing. This requires a lot of fill rate, which is not very problematic if a simple lighting model is used, with relatively few computations and texture fetches per fragment, but as the fragment programs complexity grows due to the use of more advanced lighting models, the fill rate requirements becomes a problem. To reduce the problem, techniques were introduced that would do an early z-pass where only the depth value of the objects would be written. Then the scene would be rendered as usual, ensuring that only those objects passing the depth test would be shaded.

Whilst this reduces the number of fragments shaded to mostly those contributing to the final result, filling the depth buffer also requires a lot of fill rate.

2.3.2 Deferred Shading

Another rendering model that has become quite popular is deferred shading. Unlike forward rendering, it only process the scenes geometry once and then later apply lighting by rendering the light contributions as geometry in a subsequent pass. This allows for more lights, without the enormous fill rate requirements.

When rasterising the geometry, instead of calculating the colour and light of the closest fragment, the geometric and material properties necessary for such calculations are stored to a g-buffer. The lights are then rendered into the scene as geometric shapes, either as full screen

quads, or as more optimised geometric objects corresponding to the shape of the light's influence region, e.g. a cone for a spotlight. Each fragment within the light has then the light contribution computed, using the values from the g-buffer, and is added to the final result.

The **g-buffer**, or geometric buffer, is a name given to a buffer comprising of one or more textures used to store geometric information. It is commonly used in deferred shading, where several textures are used to store, in one form or another, the position and normal of the fragment as well as material information needed for the later shader stages. It uses what is known as **multiple render targets**, or MRTs, which means that a fragment shader can output values into several buffers in one invocation.

The values stored in the g-buffer normally usually includes the position and normal vector of the fragment, as well as material properties needed for the shading model used. It is also common to compact these values, so as to reduce the size of the g-buffer and the number of texture reads needed for retrieval when applying the lights. Instead of storing the position in full precision, it is common to store the Z-depth, and reconstruct the world space coordinates from the screen coordinates and the stored depth. For the normal there are several ways in which it can be compressed into two values, see Pranckevičius [12].

2.3.3 Forward vs Deferred

Since deferred shading only need to process the geometry of the scene once, and can render the lights in a way that only shades the effected fragments, its use is spreading rapidly. But it does have its drawbacks. Since the g-buffer is comprised of several textures, in addition to those needed by the lighting model, which needs to be read for each fragment being shaded, the amount of texture reads can become prohibitive. And the size requirements of the g-buffer can become quite large, because of all the information which needs to be stored. Nor does deferred shading support transparent objects, since the g-buffer only stores information about the closest fragment. This has lead many to use a mix of deferred and forward shading, where opaque objects are handled by the deferred shading, and then the transparent objects are rendered in using forward rendering, which leads to a more complex render engine. Nor does deferred shading automatically support MSAA, unlike a forward renderer, but support for MSAA can be explicitly added.

There is also a third rendering model that attempts to reduce the fill rate requirements of a forward renderer as well as the memory requirements of a deferred renderer, which is called the "Light pre-pass model". This a very recent addition introduced by Wolfgang Engel [4], which should also be possible to use when accelerating photon mapping on the GPU.

2.3.4 Ray tracing

A popular alternative to rasterisation, often used for computer generated images, is ray tracing. Instead of transforming the data into screen space, rays are randomly shot from an eye position and into the scene, reversing the path of the light hitting the eye through the scene. When the ray intersects an object, the material properties and light contribution is calculated to find the colour of that position.

To check if the position is in shadow, so called shadow rays are generated and shot toward the lights. If there are any intersections closer to the fragment than the light, the light is occluded, otherwise calculate the lights contribution.

For reflections and refractions, one can simply continue to trace rays into the scene based on the incoming direction of the previous ray. There are different methods to determine how long a ray should be traced into the scene after the first intersection. Some choose to use a cut-off limit based on how much the ray will contribute to the final colour of the first intersection, whereas others use Russian roulette to determine if the ray should reflect, refract or be absorbed.

The benefits of ray tracing is that it is easy to compute shadows and reflections, which is very hard to do using rasterisation. Unfortunately, to get good results many rays needs to be traced, making it a very costly technique. As a result photon mapping was created as a supplement, so ray tracing would be used for direct lighting, reflections and refractions, and photon mapping for the indirect lighting, creating a good result faster than just by using ray tracing.

Since tracing photons through a scene is very similar to ray tracing a scene, difference mostly being the where they are shot from, we recommend following the research being done on ray tracing. Alternatively one could simply make use of NVIDIAs Optix [10] engine to trace the photons.

The *k***d-tree** is a binary tree structure used to partition along different axis. A common way to create the hierarchy is to split along the x-axis first, then the y-axis and lastly the z-axis, and the repeating the pattern on the downward path. Where split the current plane is commonly done by taking the median of the entries in that space, though when generated before the position of all the entries are known, it can be arbitrarily chosen. In such cases, it is important to balance the tree afterwards, to make sampling it as efficient as possible.

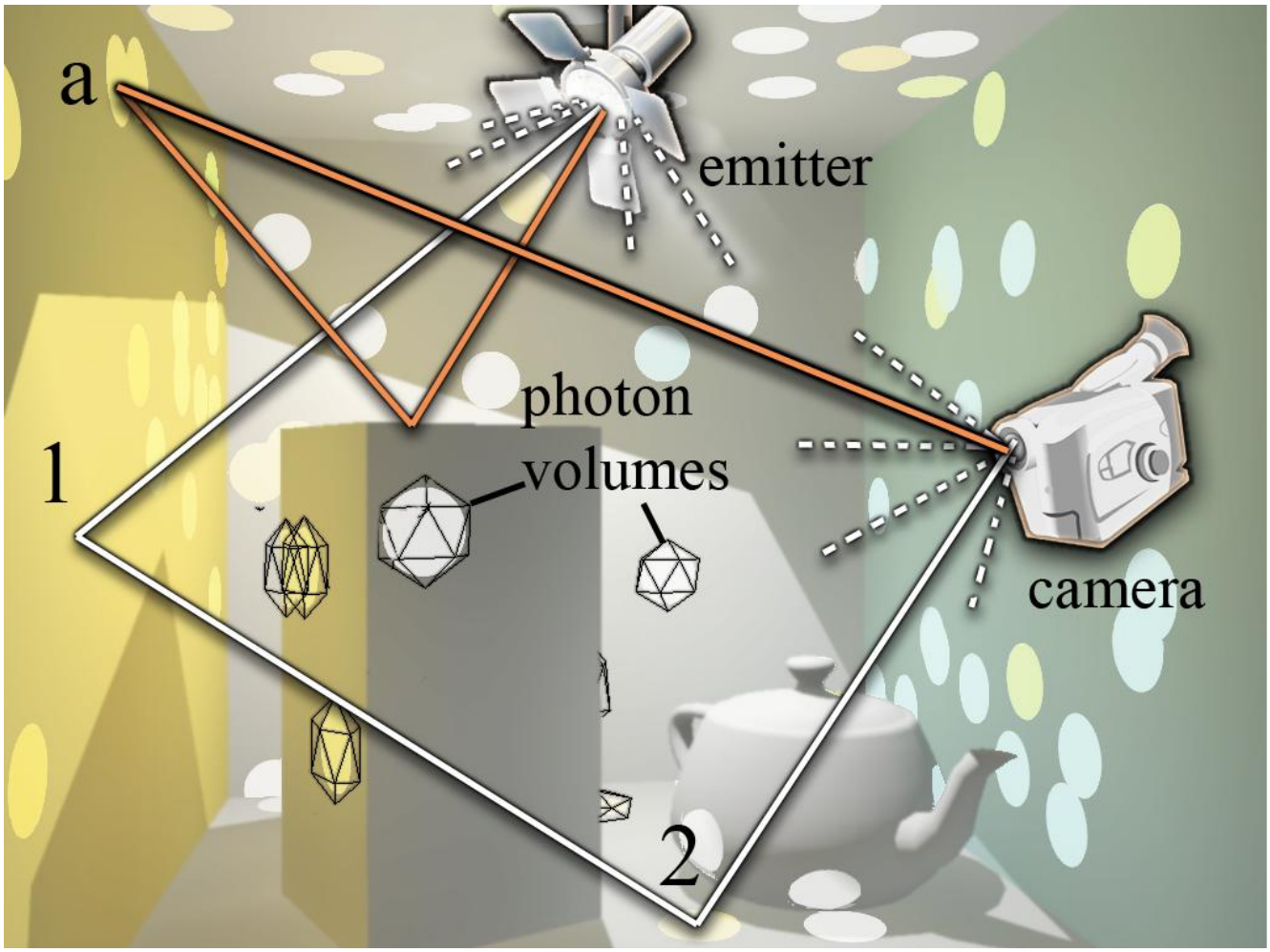


Figure 2.1: Figure showing photons bouncing from the light into the scene before being captured by the camera. Taken from Luebke[8]

2.4 Brief overview of Photon Mapping

Photon Mapping is an algorithm that solves the problem of calculating indirect lighting. For each light in the scene n photons are sent out into the scene, tracing them until they are either absorbed, exits the scene, or hit a cut-off limit. The cut-off is commonly a predetermined max number of bounces, minimum contribution of light or determined by Russian roulette. At each intersection only the relevant information of the photon is stored into what is called a photon map, with the position used as coordinates. This map is then sampled during rendering, often called the gathering step, for the m closest photons to the pixel being shaded.

The underlying data structure of the photon map, when implementing on the CPU, is commonly a ***k*d-tree**. This is a binary tree structure that partitions the space along different axis. A common way to create the hierarchy is to split along the x-axis first, then the y-axis and lastly the z-axis, and the repeating the pattern on the downward path. Where split the current plane is commonly done by taking the median of the entries in that space, though when generated before the position of all the entries are known, it can be arbitrarily chosen. In such cases, it is important to balance the tree afterwards, to make sampling it as efficient as possible.

The algorithm is often implemented in the following way: For each light in the scene, n photons are emitted into the scene in a random direction on the hemisphere of the point of the light. These photons are then traced through the scene, and stored in the photon map. What is stored when depends on the solution being implemented. A common way is to use Monte Carlo method to decide if a photon is absorbed or continues to bounce through the scene, though one can also record each surface the photon hits and decrease its intensity based on the material of the surfaces until it hits a cut-off limit. Regardless of which way the photons are traced, the hits are stored in the photon map with the coordinates of the intersection used as an index into the kd-tree.

The resultant photon map is then balanced and sampled during the final rendering of the scene, to estimate the radiance at the point being shaded. This entails finding the m nearest contributing photons to the point, adding their contribution to the final result. This contribution is then used by a filter kernel to reconstruct the illumination. There are several filters that can be used, a basic one being a disk where $\kappa(\overrightarrow{d}) = \frac{1}{\pi r^2}$, and a 3D Gaussian filter with a distance from the mean of $|\overrightarrow{d}|$ is a more advanced alternative. Here $\overrightarrow{d}$ is the distance of the photon from the surface being shaded. The size of the filter kernel varies from position to position based on how large a radius was needed to find the m nearest photons in the kd-tree.

Photon Mapping has traditionally been used for offline rendering done on the CPU, using ray tracing to trace the photons through the scene, storing the results in a kd-tree since this allows for quick access during the gathering step. To compute the photon map each photon is traced individually, as their paths quickly diverges, and thereby resulting in a lot of intersection testing which is costly to compute. However, even though the photons are costly to trace, it is often used to speed up ray tracing a scene, as it is cheaper than increasing the number of primary rays to achieve the same result. The ray tracer then calculates the direct light contribution and the photon map is used for the indirect light contribution.

2.5 Photon Mapping on Programmable Graphics Hardware

Purcell et al [14] shows how to accelerate Photon Mapping by offloading it to the GPU entirely. Being an early GPGPU implementation from 2003, it is limited not only by the nature of the GPU, many simple stream processors in parallel, but also by the need to fit the algorithm into a graphics rendering programming model.

The major difference from regular PM is that instead of using a kd-tree, which has a random memory access pattern, to store the photon data a different type of data structure is used which is better suited to GPUs constraints. The photons are stored in a uniform grid with the grid position determined by one of two functions; Bitonic Merge Sort run in a fragment shader or Stencil Routing in a vertex shader.

2.5.1 Photon tracing

They trace the photons through the scene by using a full screen quad and a fragment shader, taking the photon positions and directions and returning a texture containing the resultant photons positions. The photons are traced through the scene by using the results from the previous

frame as input for the current, storing the final photon positions into a separate photon map. For the full details on the fragment program ray tracer they base their implementation on, see Purcell et all [13].

2.5.2 Bitonic Merge Sort

The Bitonic Merge Sort is a fragment based method that index the photons through sorting them by cell and then finding the index of the first photon through binary search. This allows for an arbitrary number of photons in any given position, and gives a compact grid based photon map. Bitonic Sort is a so called a sorting network, since the comparisons can be done independently of each other. This makes it an ideal sorting algorithm to use on GPUs, as each thread can be run independently. The data is iteratively split into smaller lists until there are only two objects in each list. Each list is the sorted and then merged with another list. These lists are then sorted before again being merged with another list. This continues until there is only one sorted list left. A more detailed explanation of Bitonic Search, along with source code and examples, can be found in Fernando and Randima [5].

The cost for sorting the photons, $O(\log_2^2 n)$ steps needed by n processors to sort n elements, can be very high. With $n = 1024^2$, it would require 210 passes. On the other hand, accessing the stored photons costs $O(\log_2 n)$, and can be done in a single pass.

2.5.3 Stencil Routing

Because of the high cost of Bitonic Merge Sort, Purcell et al [14] gives an alternative single pass solution they call Stencil Routing. This works with a vertex shader, making use of the stencil buffer and the ability to write a point to an arbitrary position in a buffer. This speed increase comes at the cost of only being able to store a limited number of photons in any position. It also wastes space where less than the limit of photons are stored. However it allows for redistribution of power between photons stored at a given position, so one can scale the power amongst those already stored by those overflowing to simulate the their contribution.

The grid is allocated with cells of size g^2, allowing storage of $mg \times g$ photons per grid. Each photon is the drawn as a glPoint of size g, then the stencil buffer is used to control which pixel the photon is written to. This is done by filling the stencil buffer with the values of 0 to $g^2 - 1$, and having the stencil test to write on equal, then increment, allowing only one fragment at a time to pass through. Since the stencil buffer records how many photons have hit a certain cell, the stored photons can be scaled with the estimated power of the overflowing photons. Although this then assumes that the stored photons has a similar colour combination as the overflowed photons.

2.5.4 Radiance Estimation

Because this technique uses an ordered grid to store the photons it can not use the regular way of gathering photons, so the authors have instead come up with an alternative algorithm they call the kNN-grid method. This searches from the point being shaded out to a predetermined maximum radius. It starts by checking with a small search radius around the point being shaded,

Figure 2.2: Render from McGuire and Luebkes [9] solution.

which may cover parts of several cells. For each cell covered, the photons within are checked to see if they too are within the current radius, and if they are the energy is added to the radiance contribution. The search radius is expanded until the predetermined number of photons has been found. After which, those cells covered by the radius but not yet searched are processed and the applicable photons within are added to the radiance estimation. This makes the results deterministic, removing any randomness to the selection of photons, which could have led to artifacts.

2.6 Hardware-Accelerated Global Illumination by Image Space Photon Mapping

McGuire and Luebke [9] analyses the Photon Mapping algorithm, focusing on finding compute expensive parts that can be made parallel to be run on the GPU. Through their analysis they find that the initial step, emitting photons and calculating the first intersection is very expensive since it requires sending out a large amount of photons. And since they all have a common point of origin in the emitting light, it can be calculated efficiently using the dedicated rasterisation hardware on the GPU. This is similar to normal graphics rendering, except that instead of storing the RGB value of the pixel to the frame buffer, it stores the relevant data in what they call a Bounce Map. See Table 2.1 for detailed information on what the Bounce Map contains.

Furthermore they found that the sampling of the photon map in the gather step shares the same characteristic, each sample ray is sent from the same point. However, since the pho-

Definition	Bounce Map	Photon Map
Φ_o Power in watts	$float[3]$	$float[3]$
ρ_ρ Path density estimate	$(\lvert\overrightarrow{\omega_o}\rvert - 1)$	$(\lvert\overrightarrow{\omega_i}\rvert - 1)$
η_i Index of refraction	$float[1]$	–
$\overrightarrow{x}$ Photon position	$float[1]$	$float[3]$
$\overrightarrow{\omega_o}$ Incident light vector	$float[3]$	$float[3]$
$\overrightarrow{n_p}$ Normal to surface hit	–	$float[3]$
floats required	8	12

Table 2.1: Storage requirements for bounce- and photon-maps

tons does not have any mass, rasterising them directly into the scene will not give the desired result. Instead they suggest that the photons should be given an extent and "splatted", see Stürzlinger et.al. [15], into the scene instead, blurring the resultant "photon volumes" into the scene. This then again takes advantage of the rasterisation hardware on the GPU, rendering those photon values visible from the eye into the scene, applying a special program which adds the contribution from the photon to the underlying pixel.

In addition, they identified a problematic area where Photon Mapping, when filtering, could sample photons which where obstructed from the point being shaded. Where earlier Photon Mapping approaches fixed this by using different radii on the filter kernel, ISPM solves it by compressing the kernel along the normal of the surface being hit by the photon.

2.6.1 Photon tracing

As outlined in the beginning of section 2.4 a normal way of tracing the photons are one by one, then doing intersection tests either with the whole scene, or the closest area found in the scene-graph, before storing the results in a *k*d-tree. With ISPM, the first step is done on the GPU, taking advantage of the specialised hardware and its parallel nature. In addition, it fits in nicely in a regular graphics rendering pipeline using shadow maps, as both the shadow map and the bounce map can be generated at the same time by outputting the depth to the shadow map and the data specified in table 2.1 to the bounce map. The bounce map is then transferred back to the CPU where a ray tracer continues to trace the photons, in world space, until a termination criteria is met. At each bounce, they store not only the values normally found in a photon map, but also the unit normal $\hat{\eta}_p$ of the surface being hit, as well as a coarse estimate ρ_p of the probability that photon have scattered this many times.

2.6.2 Radiance Estimation

Normally, photons are gathered, that is sampled, from the *k*d-tree based photon map to find photons within a given area. McGuire and Luebke [9] use another approach called splatting. This means the photons are converted to polygons, called Photon Volumes, and then splatted into the scene. This approach allows them to skip the expensive part of balancing the photon map, as well as doing look ups into the map to find which photons are in a given area for each

fragment being shaded. It also means they can make use of the rasteriser hardware, allowing for the photons to be splatted in parallel.

However, by splatting the photons into the scene they say they forgo the variable sized filter kernel, and choose instead to use a 1D falloff function mapped onto a symmetrical 3D ellipsoid which is suppressed along the normal of the surface. This allows them to implement it as a texture fetch from a 1D texture, calculating the texture coordinate as a distance between the point being shaded and the photon affecting it. The values in the 1D texture will have to be tweaked on a scene by scene basis to make sure it gives a reasonable trade off between noise and aliasing in the indirect light.

In the next Chapter, 3, we will show approaches on how to calculate a variable sized filter kernel for use with this solution.

Chapter 3

Method

To achieve the best results with photon mapping it is necessary to vary with width of the filter used to blur photons into the scene. A fixed width filter can be used, but at the cost of increased noise, blurring or a combination of both.

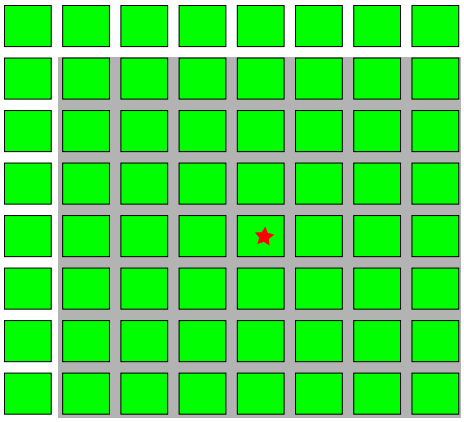
To find a variable size radius for the filter pass, there are several approaches one can explore. There is no simple one size fits all solution, as it depends on the underlying hardware it should run on, how other processes are using that hardware, what constraints that hardware imposes, in addition to the size of the problem being solved.

In addition to running on the GPU, we want the solutions to be implemented in an OpenGL graphics context. The reason for this is that it should be easy to fit into existing pipelines by not adding any extra context switches, nor be tied to any platform specific API. It could have been done in a CUDA context, though that would limit its use to NVIDIA graphics cards. An alternative would be to use an OpenCL context, but that is a new API still maturing and code that runs well on one GPU or OS, does not necessarily run well on another.

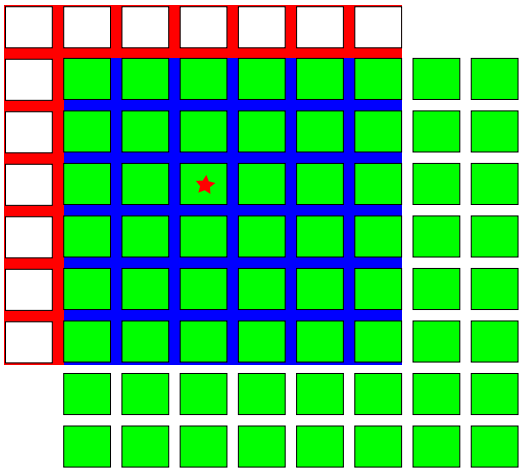
Utilising a graphics context does not guarantee that it will run well on all platforms, though the constraints it imposes should mean that it runs at comparable speeds on different GPUs. It is also considered a good practise to use a graphics context for code that is not very divergent, as the GPU makers spend a lot of resources on optimising the scheduler for non-divergent graphics workloads.

When creating a CPU implementation it is common, as mentioned in Section 2.4, to use a *k*d-tree to store the photons, and then sample that outward from an initial position until enough photons have been found. However, when implementing it on a GPU, the strength of the *k*d-tree, quickly sampling random positions, becomes a major drawback. The memory controller of a GPU is built for linear reads from memory, with neighbouring fragment accessing neighbouring memory locations, which does not correspond well with the access patterns for a *k*d-tree when sampling photons within a radius. So instead we focus on finding solutions built around the native structures of the graphics pipeline, textures, and accessing them with patterns better suited for the GPUs memory controller.

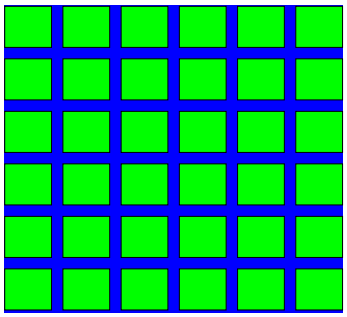
For our GPU solutions, we give as input a density map containing the number of photons for each pixel position in screen space, created by splatting the photons into the scene. The solution then returns a radius map, a texture with the dimensions and format of the input texture storing the resultant radii for each position. For simplicity we have chosen to use a box filter instead of a circular filter.



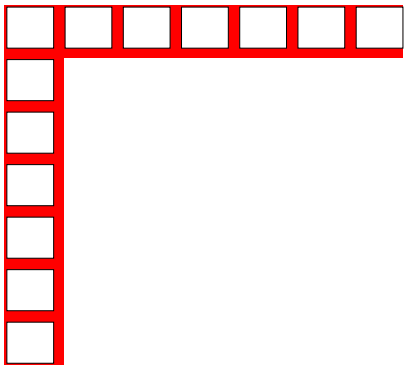
Sample area, $r = 3$



Sample area exceeds density map



Sample inside area



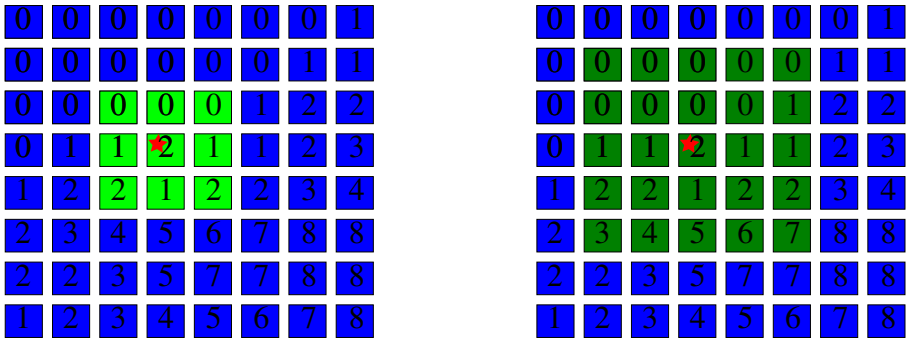
Scale by outside area to inside area, here $13/36$

Figure 3.1: How to deal with the case where the sample area exceeds the bounds of the density map.

3.1 Handling the outside area

Since the area to be sampled can easily extend outside the boundaries of the density map as the radius increases, the solutions need to be able to cope with this. One way is simply to clamp any outside values to 0, which is easy to do through clamping either in the texture sampler or shader. However, this can lead to increased radii near the edges, which might not be suitable.

As an alternative, one can make the assumption that the photon density outside the density map is similar to that within the map. This then allows scaling up the sum of photons sampled from the density map with the ratio between area outside the density map to the area inside the density map. A simple example of this can be seen in Figure 3.1.



$r = 1$, photons found 9, samples taken 9 $r = 2$, photons found 41, samples taken 25

Figure 3.2: Shows how the naive brute force solution samples a density map to find the smallest radius with at least 25 photons.

3.2 The Brute Force approach

Our brute force solution for finding the radii is similar to that employed by Purcell et.al. [14], as covered in Section 2.5.4. It loops from a radius of 1 to a predetermined maximum radius. For each radii it samples all positions covered by the radius, then checks if the sum of photons is larger than the predetermined minimum. Should enough photons not be found, the search continues by increasing the radius by one and sampling all positions within the new area. When enough photons have been found, the resulting radius is stored in the output texture and the search is terminated. To avoid extreme radii, the search will terminate if the radius exceeds a predetermined maximum radius. In which case the maximum radius is stored.

Our solution is naive with no optimisations and is heavily dependent on the texture sampling rate of the GPU since for every increase in radius the number of samples needed grows as shown in Equation 3.2, making it useless for all but the smallest n. It is mostly useful as a reference implementation, and in cases where a small memory footprint is more important than speed.

$$\sum_{r=0}^{r_{max}} \sum_{i=0}^{r-1} 8i \tag{3.1}$$

Figure 3.2 shows an example of how the brute force solution in use.

3.3 Pyramidal mipmap based solution

Our pyramidal mipmap based solution can be seen as an optimisation of the brute force approach, reducing the number of samples at the cost of a more complex solution. Creating a mipmap of the density map with a summation filter allows for sampling large areas with a single sample. However, this requires matching the sample area with the different mipmap levels and most likely splitting it up for matching with several levels.

For the creation of mipmap levels from a base texture, one can use the built-in functionality of OpenGL or one can do it manually. When using the built-in function, the mipmap levels will be built with a reduction filter, though what type of filter is not defined. Commonly this is a

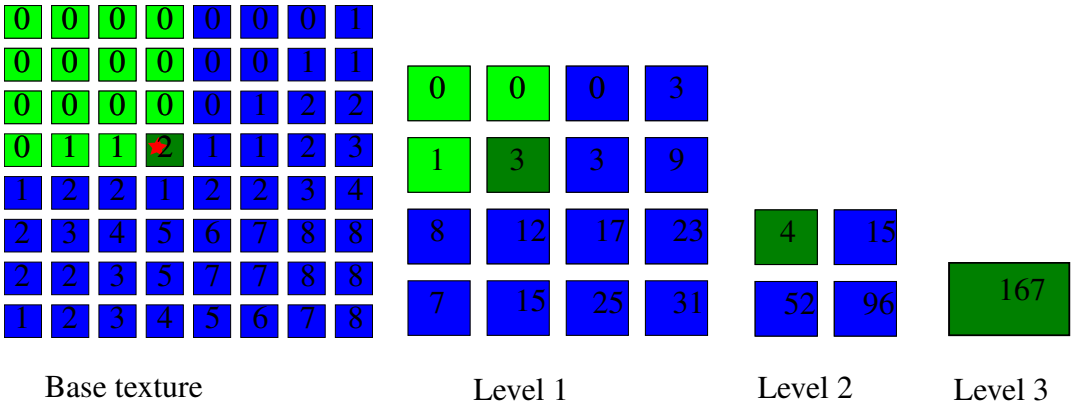


Figure 3.3: Density map with 3 mipmap levels. The highlighted areas shows how a naive solution tries to make use of the mipmap by sampling from higher levels centred around the initial position. Unfortunately this does not account for neighbouring photons on the other side of the mipmap boundary, and thus results in a much larger radius than necessary. In this example a minimum of 25 is needed, which is not found until sampling the highest mipmap level.

$2x2$ box reduction filter, but there is guarantee that not another filter is used. So when sampling from the mipmap levels, it is not certain that one can find the exact sum of the area covered. However, if a bit of variance can be allowed, multiplying the value with $(2^l)^2$, where l is the mipmap level, should give a value near the correct sum.

Alternatively the mipmaps can be generated manually, sampling the four positions from the level underneath and storing the sum in the mipmap level being generated. This requires $\log_4(n)$ passes and should be relatively quick. This also eliminates the need to multiply the sampled value when searching for the radius, and thus decreasing the complexity of the search.

We compare the speed of the two approaches in Section 5.4.

3.3.1 Naive mipmap solution

When creating a solution based around a mipmapped density map, it is easy to make the solution too naive. Allowing for a bit of variance and simply sampling upward through the mipmap levels until enough photons are found can seem like a quick and easy solution. Unfortunately this approach will most likely give a much larger radius than required, since it does not check the photons density in other cells of the mipmap. As can be seen in Figure 3.3, if quadrant of the mipmap is sparsely populated, but the quadrant next to it has a high photon density this solution will not terminate until it reaches the highest mipmap level, resulting in a radius equal to the size of the density map.

Based on the assumption that the photons are somewhat equally distributed, it is easy to construct an erroneous naive solution simply sampling the higher mipmap levels with from the initial position. If the radius should cross to another mipmap quadrant, instead of calculating what levels of that quadrant might be covered, just sample the initial quadrant, since the distribution is even. Unfortunately, due to the way photons bounce around the scene, there can easily be sharp edges where there are many photons on one side, and very few on the other. Imagine the light in the scene is hidden by a large object. Directly on the other side of the object the photon density will be quite thin compared to elsewhere in the scene. Should the edge of the

object correspond with a mipmap border, simply sampling upward in one mipmap quadrant might lead to a radius equal to half the scene, when a much smaller radius is really needed. As is seen in Figure 3.3 the quadrant being sampled is very sparsely populated. And with the naive approach, we do not find enough photons, requires 25, until reaching mipmap level 3, giving a radius of 4, where a radius of 2 finds 41 photons.

3.3.2 Our solution

So we propose a solution that crosses over mipmap quadrant boundaries. It is more much more complex than the naive solution, as it needs to match up the area to be sampled with that of the different mipmap levels. For the position whose radius we want to calculate, we clip the sample area, given by the radius centred around the point, with the top of the mipmap pyramid and downward, stopping at the first level fitting both the x- and y- axis. This allows us to find the radius of the smallest square that contains the prerequisite number of photons. After taking a sample from that level, the remaining area is split and added to a stack for later processing. As an example of how this could search the same area as the naive solution see Figure 3.4.

Since this shall run on the GPU it is important to take steps to avoid divergence in the code paths. Therefore matching the area to the highest fitting level of the mipmap pyramid should be done with the least amount of branches. One solution is to do it in two steps, first matching up with the x coordinate and range, then with the matching level l_x as starting point, match up with y. This ensures that each loop is only run for the least amount of iterations possible, and thus the least amount of comparisons possible. However, this leads to two blocks where the fragments can diverge due to matching at different levels.

A better alternative is to make use of the parallel nature of the GPU, matching both coordinates at the same time in a single loop. This could lead to better occupancy, as it will allow the GPU to process 4 values in parallel instead of pairs of 2 values.

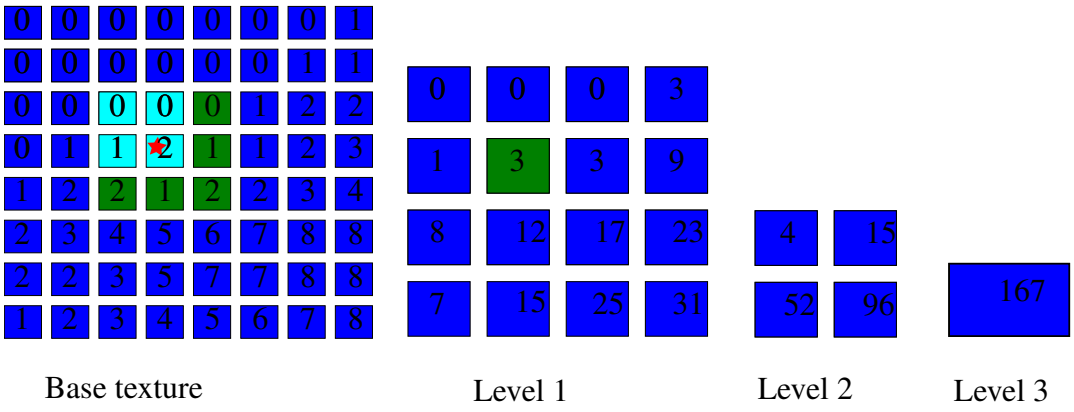
Regardless of how the match is found, we know that at the end we have found the highest level in the mipmap table and are ready to start sampling.

Our solution is to take a single sample, reasoning that the remaining area could be sampled from a higher mipmap level. This strategy requires the least amount of samples to be taken, at the cost of more iterations of the matching routine.

As each fragment most likely will match at different mipmap levels due to their coordinates, we will probably not get coalescing reads with this approach. But the code will not diverge nor wait for other fragments taking more samples.

With sampling done, the remaining area needs to be split and added to the stack. There are different ways the remaining area can be split, some better suited than others. A naive solution could be to split the area into two pieces, one above the current sample, and one for the rest. However, this approach will lead to splitting the area into fragments with a dimension of 1 on the x-axis if the y-coordinate of the fragment is odd, leading to all samples being taken from the base level of the texture. And similarly with a reversed split pattern if x is odd.

Our solution is to check if the one, or both, of the coordinates is odd, and split the area so the largest area, with even starting coordinates, are pushed on the bottom of the stack, and then the odd areas are pushed on top. See Figure 3.5 for a visual representation when first match is on the base level.

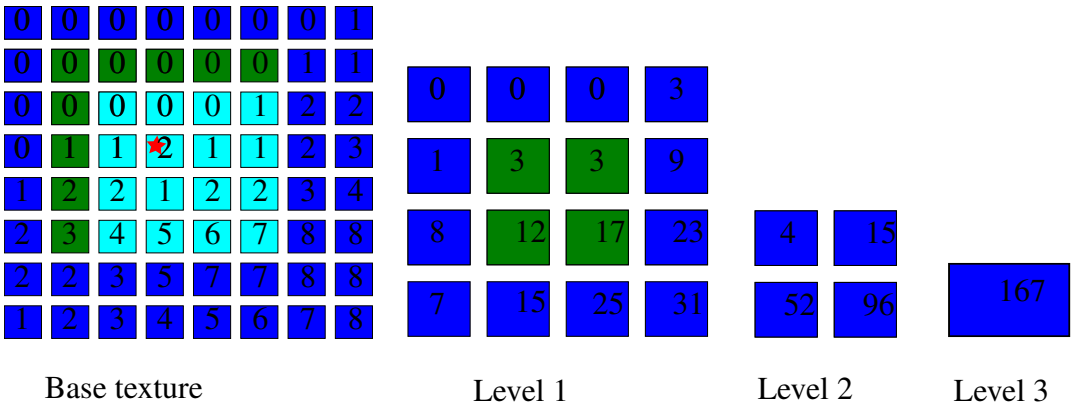


$r = 1$, photons found 9, total samples needed 6

Position to sample

Sample taken

Sampled at higher mipmap level



$r = 2$, photons found 41, samples needed 13

Figure 3.4: Shows how our solution would search for photons with the same density map as in Figure 3.3.

A benefit of doing smart splitting of the sample area helps ensure the solution does not become an overly complex brute force solution that only samples from the lowest levels of the mipmap, that could be caused by splitting the area into too small pieces.

One potential pitfall here is the use of a fixed size stack. Since, aside from DX11 compatible, most GPUs don't support dynamic stacks or linked lists we have to use a fixed sized stack. This could lead to a buffer overflow problem, when a large area is split into many smaller areas. If we accept a higher divergence, we could try to alleviate this by instead of taking just one sample from the level found, we can sample as many as will fit. Doing this will require more control statements, and possibly multiple loops running for varying number of iterations, containing, comparatively, slow texture fetches, which means more divergence and slower running time.

On problem when working with a fixed size stack is what stack size to settle on. This depends on the underlying hardware, the maximum radius that will be searched, and how the stack is split. Too large a stack, and it will be stored in global memory, which will slow the implementation down to a crawl. Too small a stack, and there will be buffer overflows as the search area becomes fragmented.

Our solution as mentioned above and presented in Figure 3.5 shows one approach that should keep the fragmentation, and thus the stack size, to a minimum.

3.4 SAT based solution

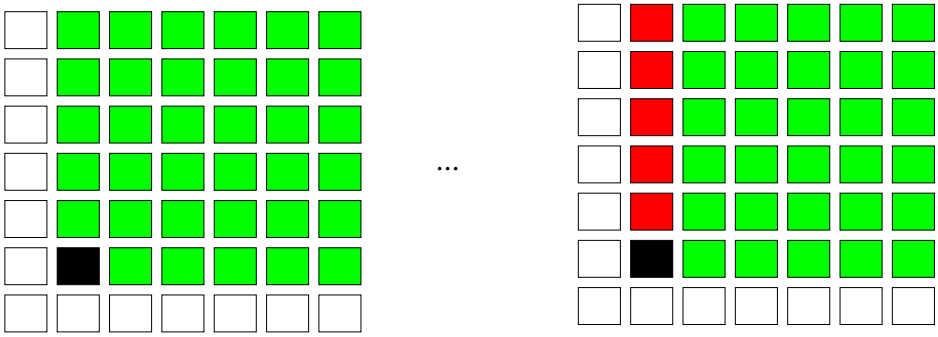
Our second solution is based upon using a summed area table (SAT). As we mentioned in Section 1.1 there are others making use of SATs for similar uses, however we believe we are the first to use them for finding a variable filter size for use with GPU accelerated photon mapping.

SATs were introduced by Crow [2] as another mipmap solution to compete with multi table mipmap pyramids. The idea was that storing the sum of intensities would allow for more general filtering options than allowed by multi table pyramidal mipmaps. Each value in the table is the result of an inclusive scan, storing the sum of all pixels from bottom left, $(0,0)$, up to, and including, itself.

A SAT can be generated in a single pass, starting at position $(0,0)$ and calculating each line horizontally. At each position, the value of the original texture is added to the previously calculated values at $p(x-1,y)$, $p(x-1,y-1)$ and $p(x,y-1)$. To sample a rectangle from a SAT also requires 4 samples to be taken. This is done by first sampling the upper right corner of the rectangle and then subtracting from it the values of the lower right and upper left corners. Then the lower left corner is added back to the sum, as it has been subtracted twice. A visual representation of this can be seen in Figure 3.6.

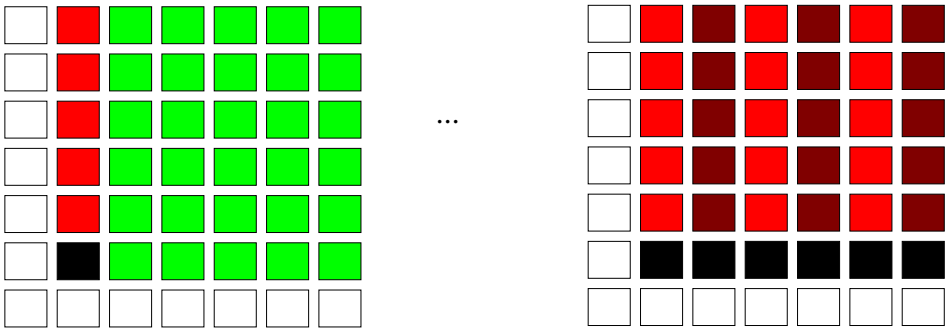
The ease with which arbitrarily sized rectangles can be sampled makes SATs a good candidate for use when finding the varying radius needed for the photon mapping.

Based on this our solution is simply to search the possible radii either through a forward search or with a binary search, sample the area given by the radius in the way outlined above and then storing the smallest radius found in the radius map.



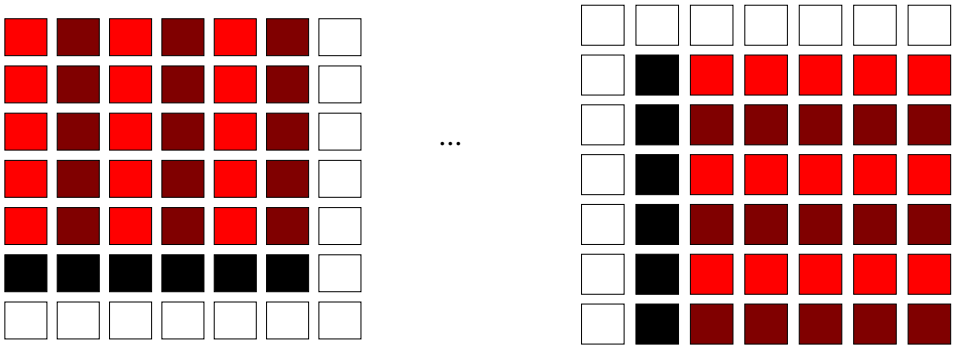
If either axis is odd, sample from base level.

Naive splitting of sample area, above and left



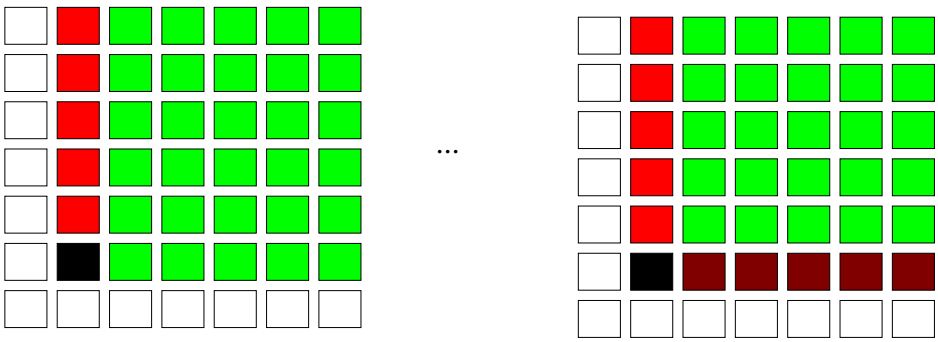
Strip of dimension 1 and odd remaining area

Result, entire area split into strips with dimension 1.



Similarly if $y-$axis is odd an splitting above.

Or if $x-$axis is odd, and splitting to the right



Solution, split off odd strips, then left with even area that matches higher mipmap levels

Figure 3.5: When splitting the sample area after taking a sample, it is important to split the area in such a way to allow for the samples at the highest possible mipmap level. Here we first see how a naive split can be done, leading to the entire area ending up as strips with a dimension of 1, and then we see a smarter solution splitting off the odd ends of the area leaving an area that matches higher mipmap levels.

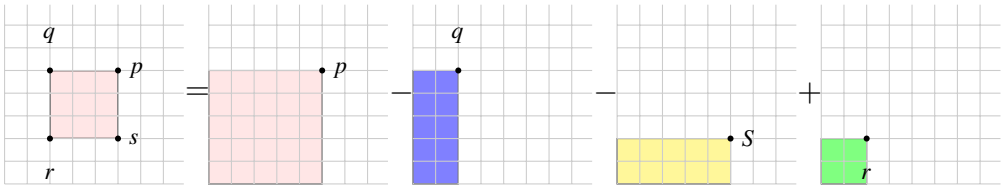


Figure 3.6: To find the sum of the area shown on the left, within p, q, r and s, we first sample the value of p. From this we subtract the sampled values of q and s before adding in the sampled value of r, since that has been subtracted twice.

Input	A	B	C	D	E	F	G	H
SAT	A	AB	ABC	A...D	A...E	A...F	A...G	A...H

Figure 3.7: Shows the input and the resulting 1-dimensional SAT

3.4.1 SAT generation

However, a speedier way to generate the SAT is needed, as the algorithm to generate a SAT given above is prohibitively expensive and not suited for GPU implementation. Hensley et al. [6] proposes an algorithm similar to recursive doubling for generating SATs in $O(\log n)$ time. They also describe ways to alleviate precision problems which can arise when using SATs, but that should not be a problem when using it with 32-bit values. Instead of calculating the SAT in a single pass into a single buffer, they split the generation up into two steps, one horizontally and one vertically, with each step consisting of multiple passes utilising two buffers. The number of passes determined in each step is determined by the length of the buffer, in the current direction, and how many samples are processed at each pass. The technique is predicated on each entry in the table being processed by a separate processor.

The pseudo-code for each processor during the horizontal step can be given as

$$buf_{output}(x,y) = buf_{input}(x,y) + buf_{input}(x - 1 \cdot 2^{pass}, y);$$

with buf_{input} and buf_{output} being switched between passes. Each subsequent pass need only do calculations for entries that have yet to be completed.

When generating the SAT it is also important to note that if the input texture is used as one of the buffers for the generation it will be corrupted. Alternatively one can use an extra buffer, which has to be populated as a step 0 as otherwise the resulting SAT will be filled with incorrect results. An example of this can be seen in Figure 3.8.

3.5 Search

To traverse the search space there are several search functions that can be used and we have chosen to focus on iterative forward and binary search.

An iterative forward search starting with a radius of 1 and expanding it by 1 until we the predetermined number of photons have been found, giving a worst case cost of $O(r_{max} \cdot sample\_cost)$ for the search itself. Due to the nature of the search, when the required number of

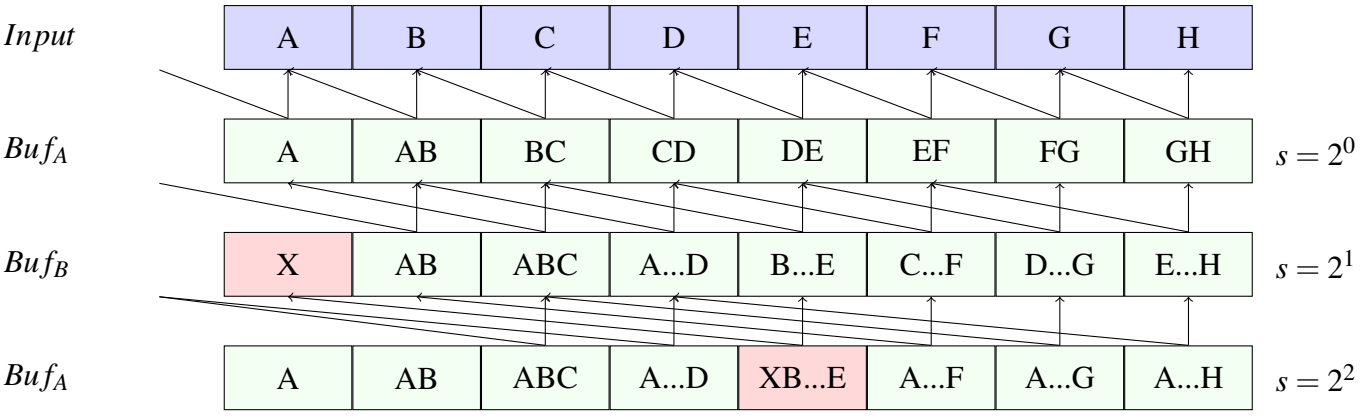


Figure 3.8: What happens when generating a SAT without making sure the first entry of the second buffer is initialised

photons have been found we can terminate the search, as we know we have found the smallest possible radius for this position.

The second search method we applied is the binary search, which when searching through the implicitly sorted radii reduces the cost down to $O(\log(r_{max}) \cdot sample\_cost)$. This could make it a good fit for our SAT and mipmap pyramid solutions which does not rely on the reuse of values from previous radii, but makes it a very bad fit for the brute force technique where to find the sum of radius r_n also includes finding the sum from r_0 to $r_n 1$. For the first radius sampled with the binary search it would have to take $(r_{(max+min)/2} + 1)^2$ samples alone, with worst case complexity of $O(\log_2(r_{max}) \cdot (r_{max}^2))$ samples.

We look closer at how the different searches perform in Section 5.6.

Chapter 4

Implementation Details

In this chapter we will go over the implementation details for the different solutions. Each of our solutions have their own section where the interesting parts of the code is listed, as well as explanations for the choices made. Since the code for handling the cases where the radius being sought exceeds the density map is the same for all three, we are explaining that in a section of its own.

The solutions were developed and tested using OpenGL 3.3 on a Ubuntu 10.04 system using a GPU from NVIDIA the GeForce GTX 260 with driver version 256.53. We also made use of Freeglut, an open source alternative to GLUT, to handle the windows, and GLEW to handle use of extensions. Even though we did not use any extensions, it was necessary to use GLEW to get a forward compatible 3.2 context.

All shaders were tested with AMDs GPU Shader Analyzer to ensure they would be runnable compatible with their Radeon HD cards from the 4XXX series up to their latest solutions.

Furthermore, whilst only implemented with OpenGL, there is nothing inherent in any of the algorithms that is tied to that specific API, so they should also work fine if implemented with DirectX or any other graphics API.

To reduce the amount of code, we only show the most relevant parts of the code. Common between them is that $\mathrm{m\_tex}$ is a single channel 32-bit floating point texture used as input, with the exact texture corresponding to the technique in use. MR is the maximum radius, commonly set to half the texture size, and MP is the minimum amount of photons needed to terminate the search.

4.1 Brute Force implementation

Our naive brute force implementation is a straight forward exhaustive search, starting with a radius of 1 and expanding outward until either enough photons have been found, or until it reaches the predetermined maximum radius. For each radius it samples all positions covered, working from left to right and top to bottom. If, after sampling the area, enough photons are found, the current radius is stored to the radius map and the search is ended.

```
float getFValuesInSquare(vec2 coords, int r){
  for(int i = -r; i <= r; ++i){
    for(int j = -r; j <=r; ++j){
      sum += texelFetch(m_tex, ivec2(coords.x + j, coords.y + i), 0).r;
    }
  }
  return sum;
}
```

As can be seen from the code, we have no checks to see if the coordinate is inside the density map, but rely instead on OpenGL clamping samples taken from outside the density map to the border colour which is set to 0.

This solution, whilst being naive and unoptimised, should be able to make full use of the GPU hardware, since the only divergence in the shading quad will occur when a fragment terminates due to having found the required photons.

This solution could be optimised by reusing the values from previous radii and taking multiple samples for each loop iteration. However we have chosen not to focus on that, as it would still only be usable for small n, and we are after a solution usable for as large an n as possible.

4.2 SAT based implementation

Our SAT solution, as described in section 3.4, is a two part algorithm. First we need to generate the SAT and then we can sample it to find the radii.

There are several ways to generate the SAT, and we are basing our implementation on the technique outlined by Hensley et al [6]. As described in section 3.4 this requires two buffers with which the partial results are rendered to, until the final result is ready. We do this by using two frame buffer objects (FBOs) each bound with a single texture. To avoid precision problems, and allow for the largest number of photons possible in a scene, we wanted to make use of a single 32-bit unsigned integer as the texture format. However, it would seem that this format is not fully supported with the drivers we tried, as we ran into strange problems that disappeared when switching over to using a single 32-bit float instead.

If any of the sample points fall outside of the density map, this will be clamped in the fragment shader, based on a conditional of being needed or not. In certain corner cases we are not interested in all 4 values, and using all actually gives the wrong result, meaning that we can not simply clamp the samples for the entire texture.

The actual generation of the SAT is done in two steps, horizontal and vertical, by rendering a full screen quad $2 \times \log_s(n)$ times, with s the number of samples taken per pass, alternating between the FBOs as input and output. An additional pass was used to populate one of the FBOs

to avoid overwriting the density map. This then leads to the following code, when generating the SAT using 2 samples per pass.

```
int decr = int(pow(2.f, m_pass));

if(horizontal)
  {
    coords = ivec2(gl_FragCoord.x-0.5, (gl_FragCoord.y-0.5)-decr);
  }
else
  {
    coords = ivec2((gl_FragCoord.x-0.5)-decr, gl_FragCoord.y-0.5);
  }
prev_value = texelFetch(m_tex, coords, 0).r;
o_val = prev_value + texelFetch(m_tex, gl_FragCoord.xy-0.5, 0).r;
```

This can easily be extended to take more samples by extending the code above with the following snippet for finding the previous value, in which *r* is the number of samples to take per pass.

```
for(int i = 1; i < r; ++i)
  {
    prev_value += texelFetch(m_tex,
     ivec2(coords.x - (i*pow(r, m_pass)), coords.y),
     0).r
  }
```

Though sampling the SAT is reasonably straight forward, there are some issues that require care. When the corners of the radius being sampled are outside the SAT, we only want to clamp and sample them if they are actually affecting the sum. When calculating a position near the left edge for instance, we only want to clamp the left corners if they delineate area specified by the radius. So if, for the top left coordinate the x coordinate is less than 0 and the y coordinate is larger the y_{max} deducting its clamped value will only lead to an incorrect sum. Similarly, we only want to add in the bottom left coordinate if both the values from the top left and bottom right corners have both been deducted.

Also, when sampling the bottom left corner, we need to make sure that we are sampling the correct value. This is easily done by deducting 0.5 from the coordinates for that corner.

Taking this into account we have the following code for sampling the SAT.

```
//get the max value
  int value = int(texelFetch(m_tex, clamp(orig.zw, 0, 63), 0).r);

//sample the SAT for the region, texture clamps to edge
if(ll.x > 0)
  {
    //upper left
    value -= int(texelFetch(m_tex, clamp(ivec2(ll.x, ur.y), 0, 63), 0).r);
  }

if(ll.y >0)
  {
    //lower right
    value -= int(texelFetch(m_tex, clamp(ivec2(ur.x, ll.y), 0, 63), 0).r);
  }
if(!(ll.x < 0 && ll.y < 0))
  {
    //lower left
    value += int(texelFetch(m_tex, ll, 0).r);
  }
```

The texture coordinates used to sample the SAT needed to be clamped in the fragment shader, as setting the texture parameters to $\mathrm{GL\_CLAMP\_TO\_EDGE}$ gave erroneous results. This probably increased the complexity of the shader, and is something that should be looked into if using the solution for production.

4.3 Pyramidal mipmap based implementation

To implement the our mipmap based solution there are two paths one can take. The first is to make use of the API built-in solution for generation mipmaps, in OpenGLs case $\mathrm{glGenerateMipmap()}$ which became part of the core in version 3.0. This gives a value close to, but not guaranteed to be, the average of the 4 values from the level below. Then we could simply multiply this value with the number of samples covered by the level being sampled. This approach takes advantage of the IHVs ability to optimise fixed functions like this in the drivers to make full use of the hardware.

The alternative is to generate the mipmap pyramid manually, only storing the sums without averaging. This reduces the calculations in the fragment shader when searching for the radii, at a higher initial costs.

When we implemented both we found that the automatic reduction led to larger radii than we saw with our other solutions. Our solution based on manual generation did not display this problem, and thus we decided to use manually generated summed mipmaps.

Implementing the search function is more complex, not least because of the use of a stack. In DirectX 11 MICROSOFT introduced support for linked lists in the pixel shader, but that is not currently supported by OpenGL. And when that support comes, it will most likely be limited to OpenGL 4.x compatible cards. So instead we use a fixed size stack, where each entry contains the bottom left and top right corner of the area to be searched.

An alternative that would spring to mind when implementing on the CPU is simply to use recursion, however recursion is not, at least currently, supported on the GPU.

When putting areas onto the stack, there are a couple of things to remember. Larger areas should be pushed onto the stack first, then add the smaller areas. This should help to minimise the chance of overflow, since the smaller areas will probably not fragment much and should thus be easier to finish sampling removing it from the stack.

Another important aspect as we mentioned in Section 3.3.2 is to split the sample area in a way that allows for matching with the highest mipmap level possible, so the number of texture fetches are kept to a minimum. This matching can be problematic when one, or both, of the coordinates are odd. The sample would then be taken from the base layer, with the remaining area split into two, one with a dimension of 1 in either x or y direction, and one with the remaining area. Should the remaining area still have an odd coordinate, it would still only match the base level, and this would keep repeating until the area has been split into pieces with a dimension of 1 in one direction. To avoid this problem, we can split the area into different pieces depending on their coordinates.

```
bvec2 range = bvec2( ((x_max - x_min) - sample_size) > 0,
                     ((y_max - y_min) - sample_size) > 0);

if (x & y odd)
  {
    if(all(range))
      {
        stack[++index] = ivec4(x+1, y+1, x_max, y_max);
      }
    if(range[0])
      {
        stack[++index] = ivec4(x, y+1, x+1, y_max);
      }
    if(range[1])
      {
        stack[++index] = ivec4(x+1, y, x_max, y+1);
      }
  }
else if(x odd)
  {
    if(all(range))
      {
        stack[++index] = ivec4(x+1, y, x_max, y_max);
      }
    if(range[1])
      {
        stack[++index] = ivec4(x, y+1, x+1, y_max);
      }
  }
else if(y odd)
  {
    //opposite of if(x odd)
  }
```

To match the coordinates with the highest possible mipmap level with which to sample from

we use a single loop doing the matching with all 4 parameters to take advantage of the GPUs parallel nature.

A costly mistake in our initial solution is that when splitting the area, it did not check to see which split would be result in the single largest area, leading to situations where it only became possible to samples from the base level and the first mipmap level. This was updated to the following code that works nicely.

```
//both odd, maximise search space
if(range[0] > range[1])
  {
    //x space larger than y
    if(range[0])
      {
        stack[++stack_index] = ivec4(pos.x+sample_size, pos.yzw);
      }

    if(range[1])
      {
        stack[++stack_index] = ivec4(pos.x, pos.y+sample_size,
                                     pos.x+sample_size, pos.w);
      }
  }
else
  {
    //y space larger than y
    if(range[1])
      {
        stack[++stack_index] = ivec4(pos.x, pos.y+sample_size, pos.zw);
      }

    if(range[0])
      {
        stack[++stack_index] = ivec4(pos.x+sample_size, pos.y,
                                     pos.z, pos.y+sample_size);
      }
  }
```

The matching itself is done through the modulo operation between two integer, a quite costly operation in the fragment shader. When a match for both x and y is found the position is sampled. The loop then continues until the entire area have been sampled, which corresponds with the stack becoming empty.

When selecting a size for the stack, we first decided on a size of 32, expecting that it should be small enough not to spill into global memory, and still large enough for our problem set. However, when testing this we found that a stack of size 5 is sufficient with the way we split the sample area.

We are clamping the coordinates manually in the fragment shader to avoid any problem with the stack due to the way we split the sample area.

4.4 Outside handling

There are different ways to calculate the ratio of outside area to inside area, and, again, it is important to choose a solution that is not overly complex. So instead of doing costly clipping between sample area and the density map, we don't need to know what parts are outside, only the size, we settled on finding the ratio directly. This we do by calculating the total area from the radius, then calculating the inside area from the clamped min- and max- corners. Dividing the outside area by the inside area gives us the ratio to scale the sum of sampled values.

One thing to note is the need to clamp the ratio to 0, to avoid it becoming negative due to floating point rounding errors.

Chapter 5

Benchmarks

Now it is time to see how our solutions perform. We will first show a graph on how each of the solutions perform compared to each other, and then look more at the details. We will cover generating the intermediate structures, how the different searches affect the results, as well as some comparisons between different GPUs.

5.1 Test methodology

Since we want our solutions to be usable with existing render pipelines with different constraints and capabilities, we decided to test the solutions as stand alone programs. This allows us to focus solely on the task at hand, and the results can be used to see how much it would affect the total render time of existing engines.

Because of this we are not using the common way of showing frames per second, focusing instead on how many milliseconds ms it takes to complete the task. We believe this is a better benchmark as it allows us to see the results from our solutions which might otherwise be hidden by unnecessary overhead from other tasks.

For each solution, we only time the barest minimum of operations needed to complete the task, so we do not count setting up the data structures, compiling of shaders or suchlike, since that is a one time cost paid as the programs starts. Instead we focus on changing the OpenGL state, populating intermediate data structures and finding the radius.

We then take the time it takes to run these operations by using an OpenGL query object, $\mathrm{GL\_TIME\_ELAPSED}$, running each test multiple times and then averaging the results. The fastest solutions have been run in 5 sets of 10^3 iterations to produce stable results. Slower solutions have been run for less iterations, but we tried to make sure they had enough iterations to keep variance low between the sets.

While we believe this is the best way to benchmark the solutions, the individual numbers for generation and search is often less than when summing together the results from individual tests. We are not certain of the reason for this, though we find it likely to be caused by the graphics driver optimising the render workload so that the changes to the OpenGL state is overlapping the end of the previous render call.

We test for data sets ranging in size from 64^2 up to 512^2, with some tests even up to 1024^2. After running the tests we see that the input data were perhaps a bit too dense, since the results

has large areas of very small radii. However, there was not enough time to generate new input data and test it, so that will have to be left for future work.

The machine we tested the solutions on is the same as used for implementation, with full specs in the start of Chapter 4. We tested with 2 graphics cards, the GeForce GTX 260 and the GeForce GTX 480 using driver version 256.53. We wanted to test with AMD FIRESTREAM 9250, however there seems to be a difference with the coordinates so we decided against it.

We will only present graphs of the results here, with a discussion of them in Chapter 6. The data is presented as tables in Appendix A.

5.2 Overall results

Looking at the results in Figure 5.1 there is no surprise to see that our SAT based solutions are the fastest. The naive brute force solution is as expected slow, but not unusable, as the results are fast enough to allow for interactive if not real-time speeds. However, we had not expected our mipmap solution to be as slow as this. There are different reasons that can be behind this which we will discuss closer in Section 6.3.

Figure 5.2 shows a clearer picture of how the three solutions perform on the GTX 480. The brute force and mipmap solutions two orders of magnitude slower than our SAT solution, however their speeds still corresponds to interactive rates.

5.3 Brute force results

Figureẽffig:brute-search shows the performance of our brute force solutions on the two tested cards. It shows linear growth in the time it takes to find the radii as the size of the density maps grows. The non-linear growth from 64 to 128 for the GTX 480 could be caused by both density maps fitting completely in the texture cache.

5.4 Mipmap results

To generate our summed mipmap pyramid is a quite quick operation, as can be seen in Figure 5.4. We decided against benchmarking the time it takes for OpenGL takes to generate the mipmaps, as various implementations will use different reduction filters giving varying results making them unsuited for use with our mipmap solution.

The difference between the time it takes to complete the search when using a forward search, as seen in Figure 5.5, and a binary search, Figure 5.6, is quite large for the GTX 260. This is probably due to the forward search terminating earlier than the binary search, as well as an effect of using integer operations. With the GTX 480 we see the difference is dramatically reduced.

In Figure 5.7 we see how our mipmap solution fares when up against the brute force solution.

For further analysis of our mipmap solutions see Section 6.3

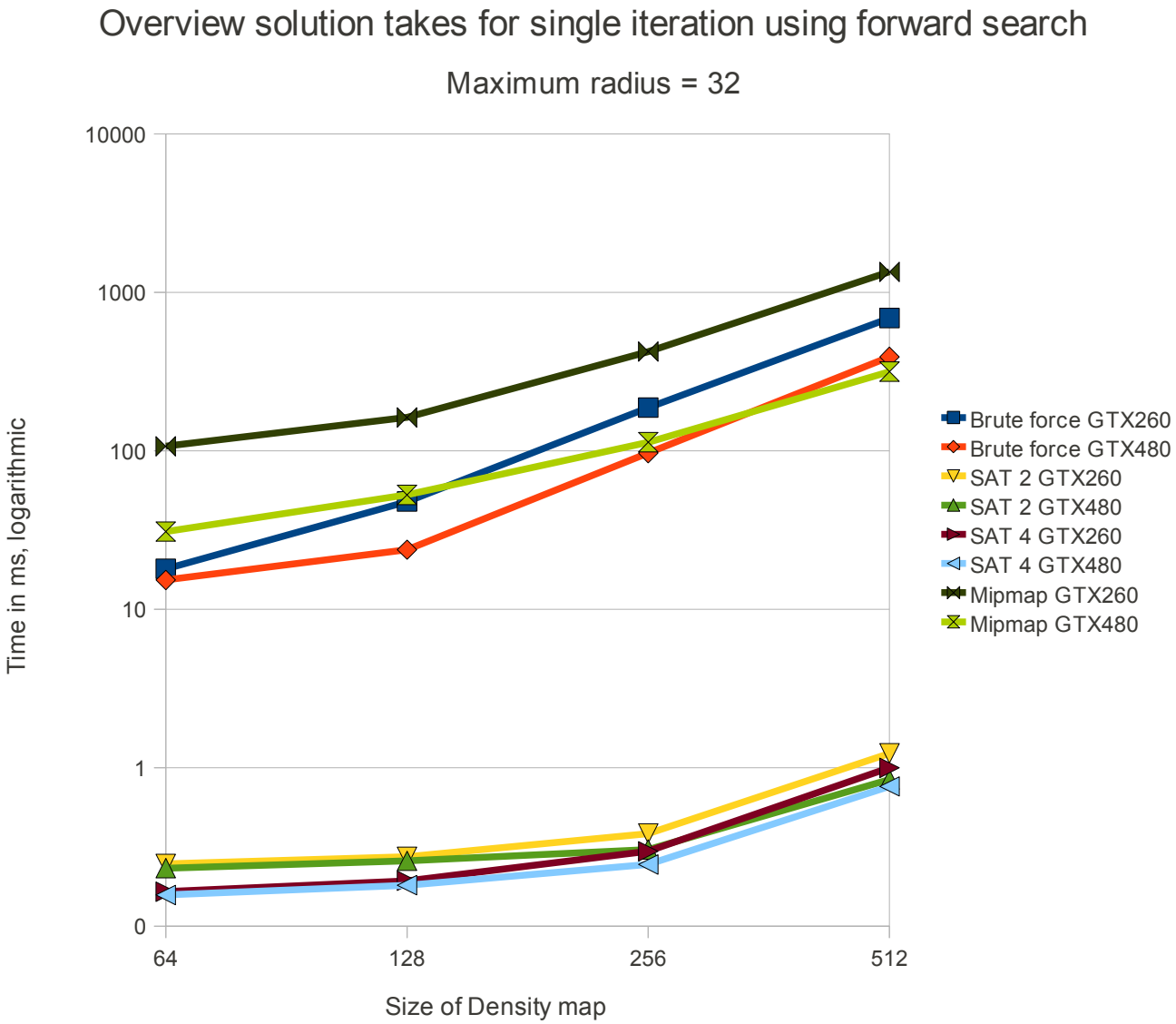


Figure 5.1: Graph showing the time it takes for each of the solutions to complete a single iteration for varying sizes of density maps.

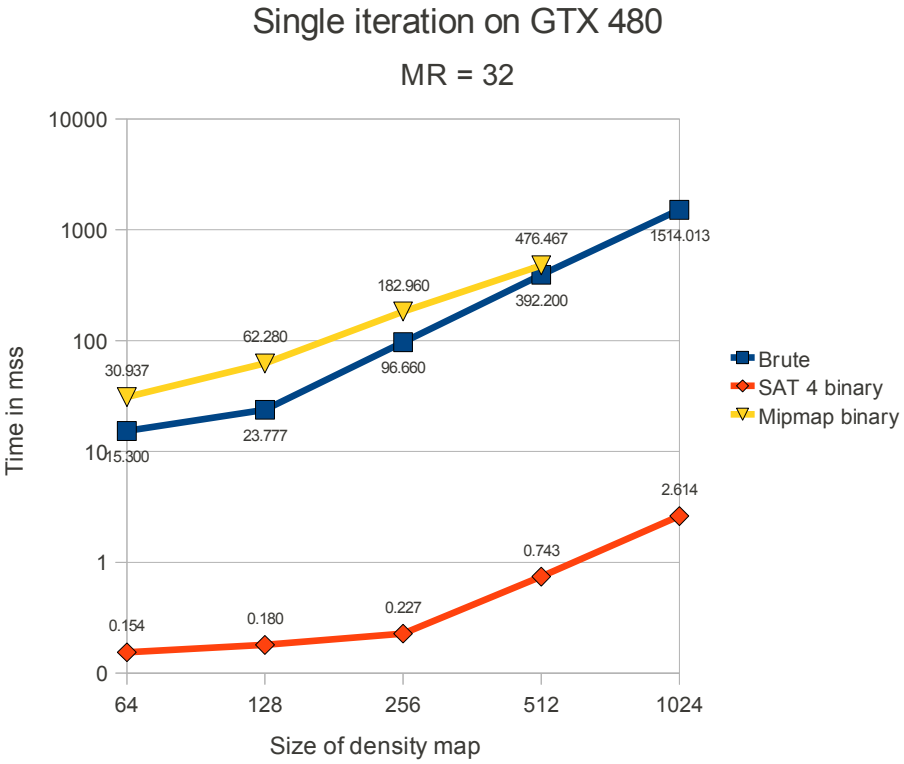


Figure 5.2: Graph showing the three different solutions running on GTX 480.

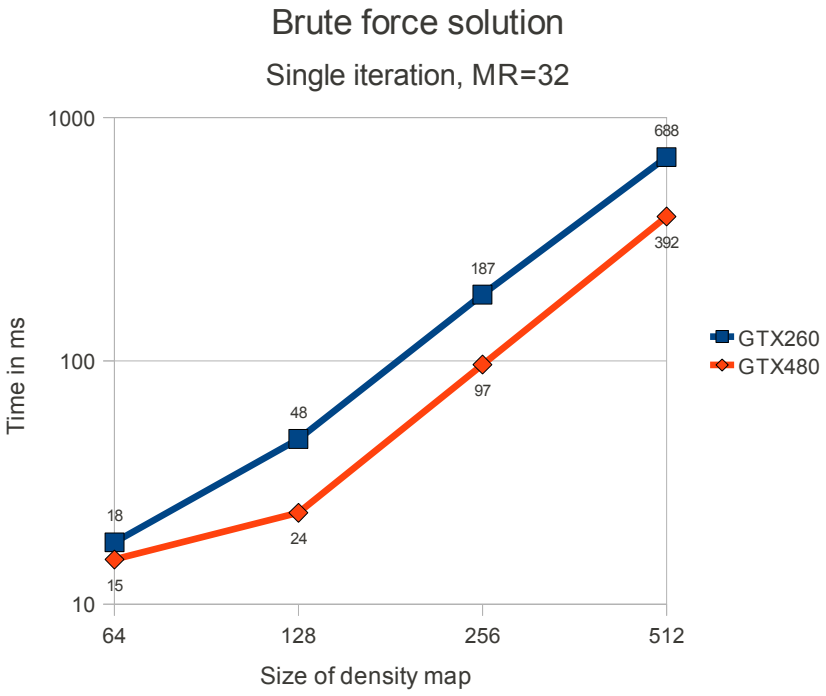


Figure 5.3: Graph showing the speed of our naive brute force solution when run on GTX 260 vs GTX 480.

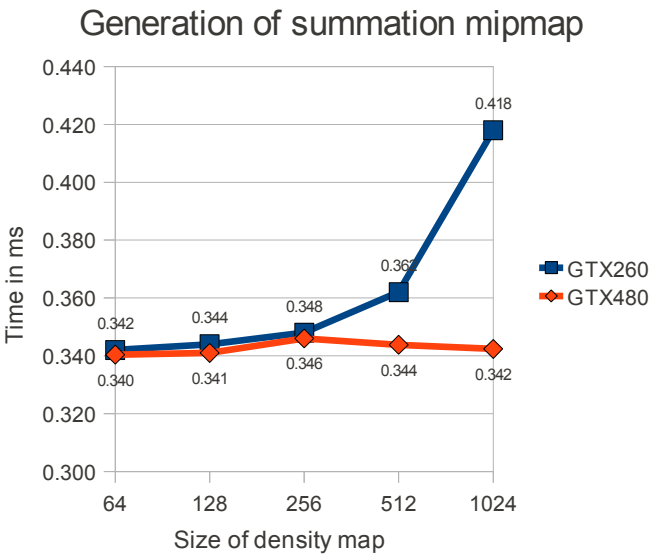


Figure 5.4: The time it takes to generate the summed mipmap pyramid.

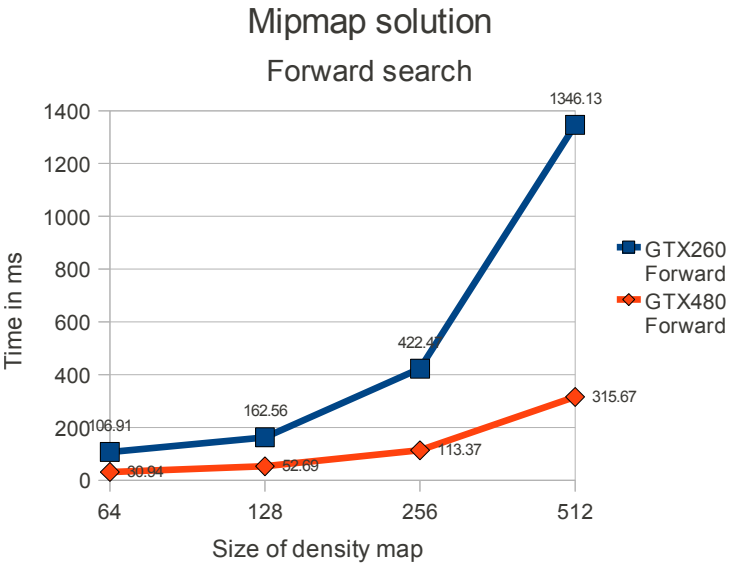


Figure 5.5: Our mipmap solution with a forward search.

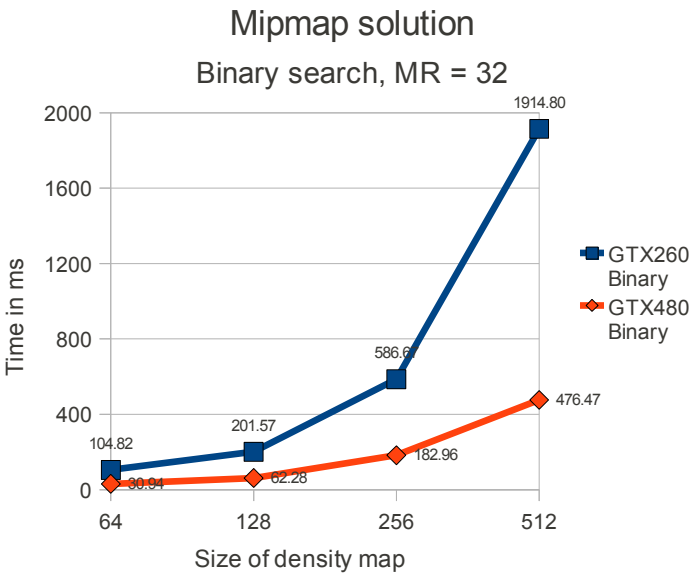


Figure 5.6: Our mipmap solution with a binary search.

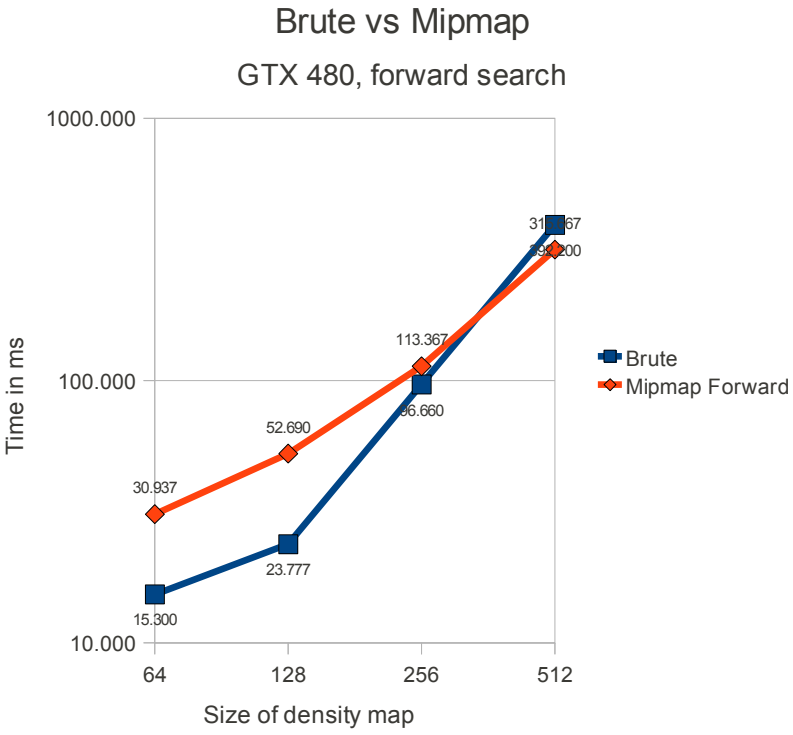


Figure 5.7: Our mipmap solution compared with the brute force solution.

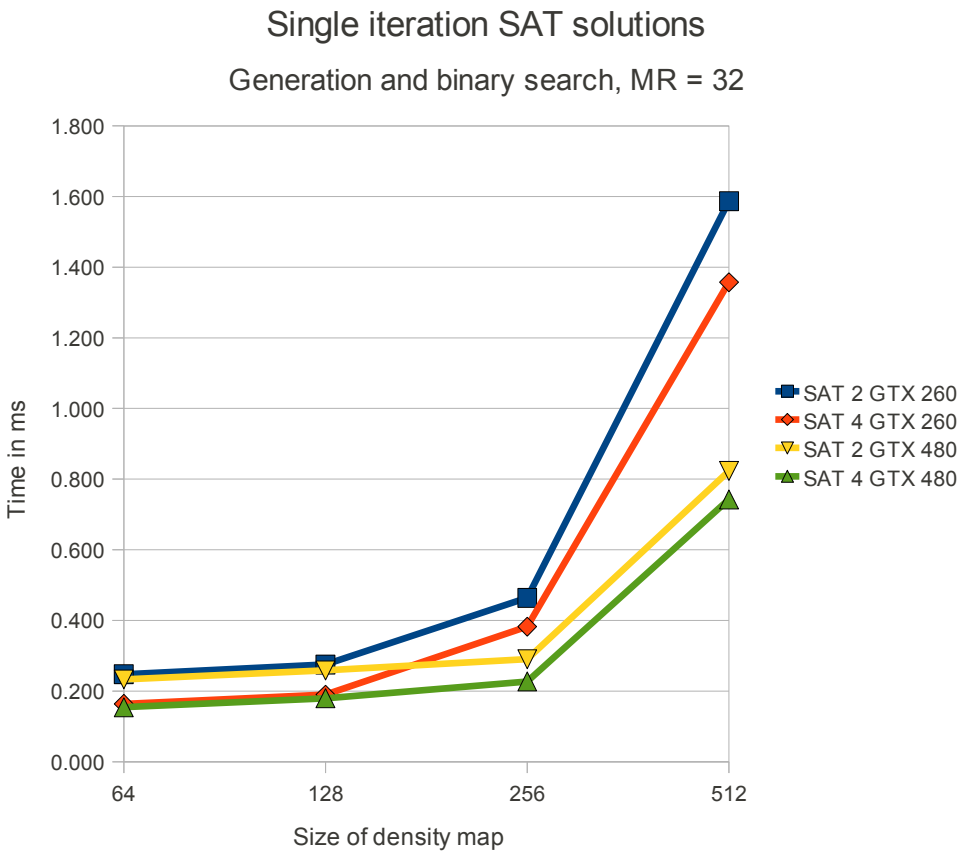


Figure 5.8: Graph showing the speed difference between taking 2 and 4 samples per pass when generating the SAT.

5.5 SAT results

Moving on to our fast SAT based solution, we see in Figure 5.8 that there is quite a large difference between taking 2 and 4 samples during the generation.

Looking closer at only the search step in Figure 5.9 we can see for the GTX 480 the binary search is faster than the forward search. However, looking at Figure 5.10 we find the opposite is true for the GTX 260.

Comparing the speed of the binary search between the two cards, as seen in Figure 5.11, we clearly see the GTX 480 outperforming the GTX 260.

Figure 5.12 shows the speed with which the GTX 480 generates the SAT when taking 2 and 4 samples. It really does make a difference.

5.6 Search speeds

As we have seen how the different solutions performed with the different searches in the previous sections, we will limit us to looking at Figure 5.13 which shows us how varying the maximum radius affects the search time of the binary search.

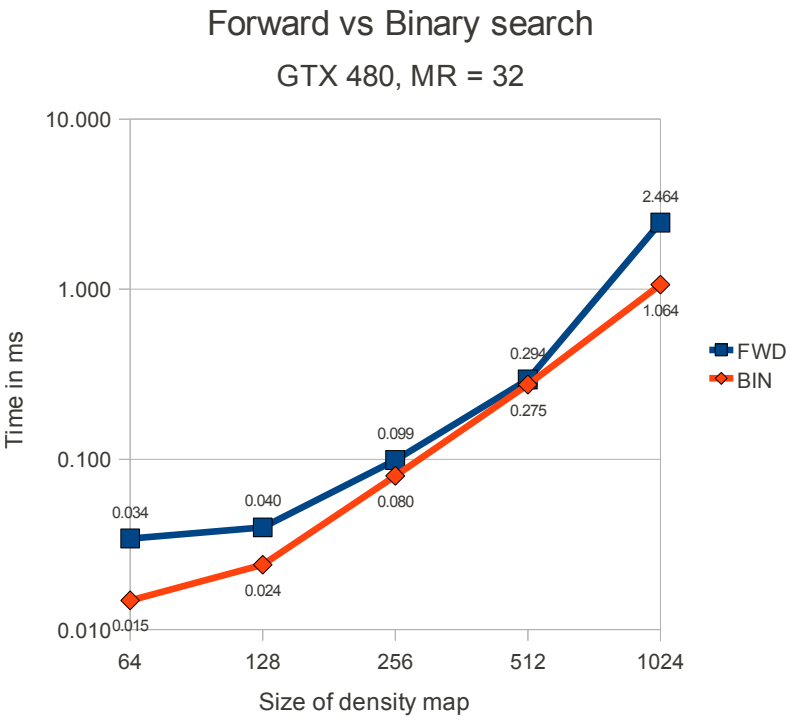


Figure 5.9: Graph showing the speed difference binary search and forward search for our input data.

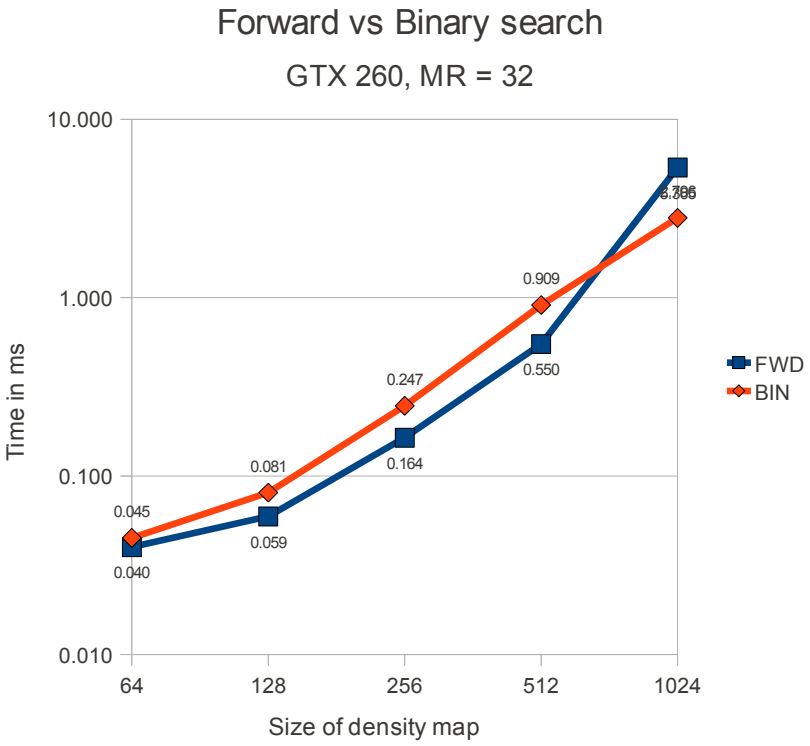


Figure 5.10: Graph showing the speed difference binary search and forward search for our input data.

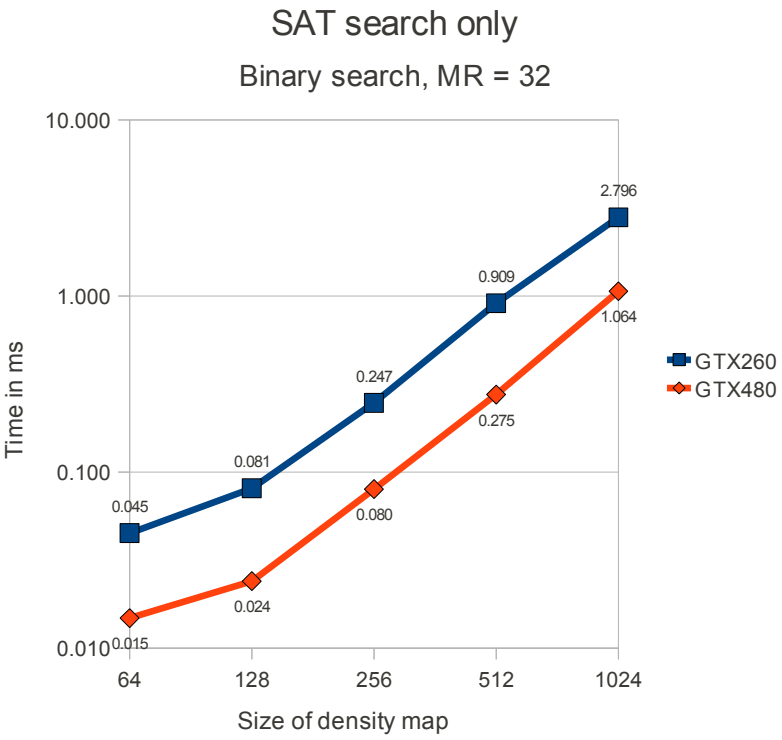


Figure 5.11: Graph showing the speed of finding the radius for a density map, excluding the time to build the SAT.

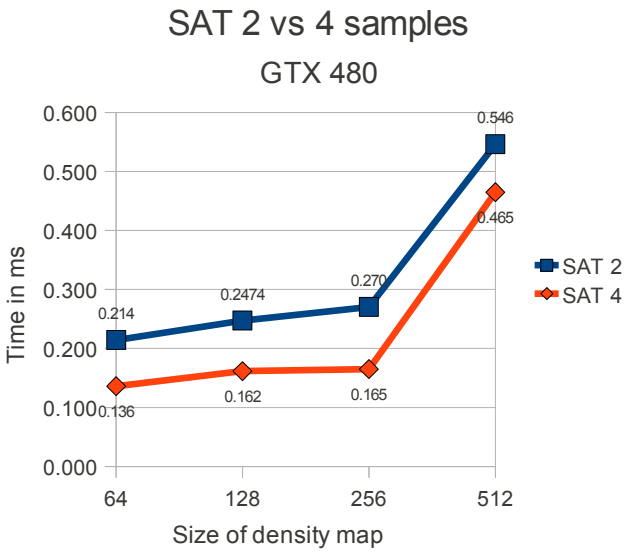


Figure 5.12: The difference between generating the SAT with 2 and 4 samples per pass.

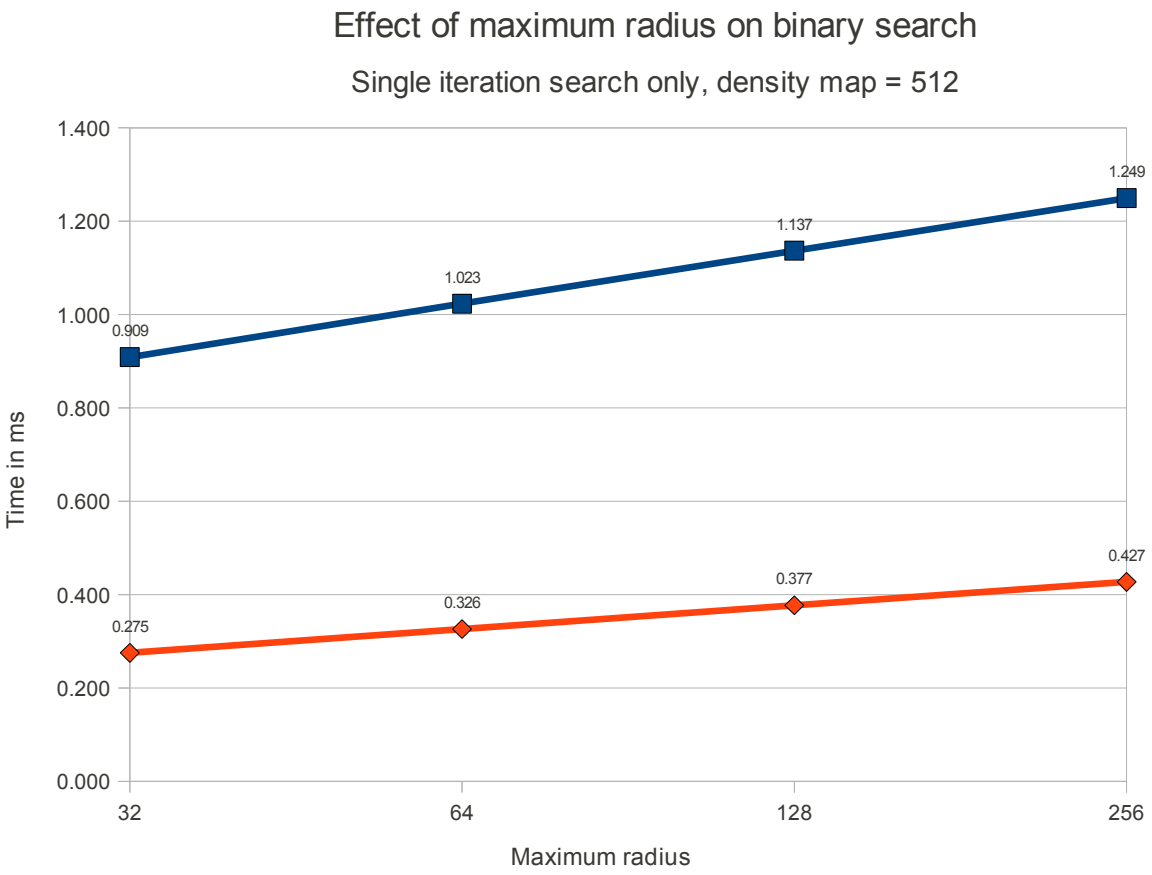


Figure 5.13: Graph showing the speed of the binary search with different maximum radius.

Chapter 6

Summaries and Discussion

It is now time to discuss the results we saw in Chapter 5. We will go into details of our solutions, seeing how the choices we made in Chapter 3 and 4 affects the results as well as try to determine other causes that could be affecting our results.

We will also give some suggestions for areas that would be interesting to investigate with future work.

6.1 Search

As both of the search types we have implemented, forward search and binary search, are common search methods we will not go much into details on them. However, it is interesting to note the difference in speed between the two types. Whilst we had expected the binary search to win hands down, it turns out that our input data has a rather dense photon population resulting in small radii. We should perhaps have tested with several data sets and varying the required number of photons, unfortunately there was not enough time.

Though this also shows how it can be useful to switch between search methods depending on the photon density.

6.2 Brute force

Turning to our brute force solution, this was intentionally made a naive solution to be a reference with which we could measure our other solutions. There are different optimisations that could be done to it that would increase its speed ranging from reusing results from previous radii through using $\mathrm{textureGather}$ operations to more optimised sampling strategies. We felt however that these optimisations, whilst surely improving the running time, would not be as interesting as alternative approaches and they were thus not explored.

6.2.1 Implementation

As we have seen in Section 4.1, our brute force implementation is based around three nested loops, with a single branch in the outer most loop to terminate search when enough photons have been found.

Each sample taken is added to the sum of the area already searched, keeping the register usage to a minimum.

For each subsequent radius to be searched, it will also re-sample the areas covered by the previous radii.

6.2.2 Pros

Because of the way the search is structured there should be no chance of divergence in the shading quad, aside from fragments terminating when they have found the required number of photons.

Due to this coherence, the texture fetches should be done in lock-step across the shader quad, increasing the chance of coalescing reads and thus allowing for faster fetching. Furthermore, as the sampling is done linear fashion, going from left to right, top to bottom, after the initial fetch, the samples should mostly be in the texture cache with the speed benefits that brings. This should also benefit other shader quads in the same region if scheduled on the same shader core.

Another positive aspect of the naive brute force approach is that it doesn't have any extra memory requirements, simply reading directly from the density map and storing the results in the radius map. This also means it is cheap to switch between shader quads as there is limited state that needs to be stored.

Lastly, due to its naive nature it should be relatively easy for the shader compiler to optimise the code compared to the more complex solutions.

6.2.3 Cons

As it is a very naive solution, it does not spend the resources available in an efficient manner. There is no reuse of results from previous radii, leading to massive, unnecessary, over sampling.

Furthermore as each sample taken is directly added to the sum of previously sampled values it is not possible to schedule multiple texture fetches at the same time, though this could possibly be optimised by the compiler.

6.2.4 Summary

All this means that the solution is reasonably good to utilise the graphics hardware when searching. Though it does not mean it is an ideal solution.

Looking at the results from Section 5.2, and in particular Figure 5.3 we see that it gives interactive speeds for density maps of size 64, 128 and 256. At these sizes it is likely that most, if not all, of the density maps fit into the texture cache of the cards.

Also, looking at the difference between the GTX 260 and the GTX 480 we see that the GTX 480 is about 50% faster than the GTX260. This could fit with the GTX 480 being able to schedule instructions from two shader quads at the same time, thereby doubling the number of texture fetches that can be scheduled at the same time. For more information on the capabilities of the GTX 480 see Voicu [16].

However, in its current form this is not really a viable solution for use with real-time graphics. Even for the smallest problem size, 64^2 with the fastest card it takes 15.3 ms to find the radii leaving 18.034 ms for the rest of the rendering operations to achieve at least 30 fps.

6.2.5 Future work

Due to the speeds it is able to achieve in its naive form, it would be interesting to see how it would perform when reusing the results from previous radii.

It could also be worthwhile to explore how the sampling could be optimised, be it taking several samples per loop-iteration or seeing how the other built-in texture operations are able to handle the workload. It is possible that by using either $\mathrm{texelFetchOffset}$ or texture gather operations could lead to more coalesced reads and reduce the number of fetches needed.

6.3 Mipmap solution

Our mipmap solution is trying to reduce the number of samples needed for each radii by making use of a pyramidal mipmap built with summation filter as an intermediate data structure. The idea being that by using the mipmap one can sample large areas by fetching a single value from the correct mipmap level.

To do this requires complex logic to match up the area to be sampled with which mipmap levels to sample from and this can then become a bottleneck.

There is also the question of what sampling strategy to employ once the correct level has been found. One could take only a single sample, thereby increasing the chance of matching with a higher level for the next iteration, or one could sample as much as possible from the level found since that will reduce the number of times the matching routine is used. Another possibility is to use a smart hybrid of the two where the number of samples taken is dependent on the area remaining after a single sample is taken. This then introduces another possibly costly step of complex logic.

6.3.1 Implementation

Here follows a quick overview of our implementation. For the full details, and some example code, please see Section 4.3. Our implementation generates the mipmap pyramid manually to get the summed values, instead of having to rely on the near averaged result from the built-in $\mathrm{glGenerateMipmap}$ function. This is due to the lack of standardised filter used for the reduction leading to larger radii than is necessary. Had it been guaranteed to be the average of the four values underneath it one could scale this up to the correct value based on what mipmap level is sampled, and thereby taking advantage of the speed with which the mipmaps is auto-generated with.

Starting with the initial area to the be sampled on the small, fixed sized, stack, a loop is started that runs until the stack becomes empty. For each iteration of the loop we match the position and area of the sample area with the different mipmap levels. With the match found a single sample is taken, and then the remaining area is split in a way to ensure the largest continuous are with even coordinates and added to the stack.

The matching is made up of two parts. First through two integer modulo operations where the x- and y-coordinates are matched up with the size of the different mipmap levels. Secondly two comparisons are done to see if the area to be sampled is equal to, or larger than, the area covered by the given mipmap level. In an attempt to optimise this, by taking advantage of the GPUs parallel nature, it uses a vector and vector operations to compare the results.

From the alternatives mentioned in Section 6.3 we chose to take only a single sample to increase the chance of matching at a higher level when the radii become large.

The remaining area after the sample is taken is then split and added to the stack as explained in Section 3.3.2.

The solution can be use both with a forward search and a binary search.

6.3.2 Pros

By using a summed pyramid mipmap as an intermediate data structure, we are able reduce the number of samples required to cover an area with radius 1 from 9 to a maximum of 5, compared with the brute force solution. As the radius grows, so does the savings. A radius of 2 will require at the most 13 samples, instead of 25.

And through smart splitting of the remaining area, it is possible to maximise the likelihood of matching up with a high mipmap level thereby decreasing the number of samples even further. Depending on the coordinates the area extended by a radius of 8 can be sampled in as little as 34 samples compared with 289 for the brute force solution.

In addition, by making use of the vector types and instructions native to the GPU we increase the chance of utilising the GPU fully.

The technique is not bound to a given search type, so can be used with the search function most suited to the distribution of the density map.

6.3.3 Cons

However, the reduction in number of samples needed to sample an area comes at the cost of a higher complexity. Since each of the fragments in the shading quad will have different starting coordinates for their areas to be sampled, they will be matching with different levels of the mipmap. This then leads to divergence between the fragments in the number of iterations needed to find the matching mipmap level, leading to idling.

By matching at different levels there is little chance that they will get coalescent reads and instead each texture fetch will be issued independently. This then greatly reduces the efficiency of the algorithm.

Another problem arising from matching at different mipmap levels is matching with values present in the texture cache. With each fragment in a quad reading different locations it less likely they will hit values already present in the texture cache. In a worst case situation it can lead to thrashing of the cache, where no texture fetch matches any of the values in the cache. Though this is not very likely to happen for smaller density maps.

There is also the matter of the type of operations done in the fragment shader. Our implementation makes extensive use of integer operations for matching up with the mipmap levels,

both modulo and comparisons, when the GPU is mainly built around doing floating point operations. The problem is then that the integer operations take more clock cycles to complete than a comparable floating point operation. This can especially be a problem with special operations like the modulo operator which can only be computed by a special part of the shader core, leading to longer waiting periods for the shading quad to complete the operation.

The use of a fixed stack increases register pressure which can increase the cost of swapping between shading quads and might even limit the possibility for dual-issue of instructions. However, as we are able to utilise a small stack, using only 5 $\mathrm{float4}$ variables this should not be a problem. However, by increasing the maximum radius the size requirements of the stack might increase.

In addition to the stack, we make extended use of general registers to store temporary variables which also increase register pressure. According to AMDs GPU Shader Analyzer we make use of 29 general registers.

6.3.4 Summary

Looking at the results in Figure 5.1, we can see that our mipmap solution is actually slower then our brute force solution. This shows that with our current implementation the reduction in samples taken is more than offset by the increased complexity of the fragment shader.

However, when we look closer at the results in Section 5.4 we can see that the method has potential. With the increased speed for integer operations as well as dual-issue capabilities the GTX 480 is capable to complete the search in about $\frac{1}{3}$ the time of a GTX 260. Part of this immense speed increase could be caused by the dual-issue of the GTX 480 as well as the beefier integer unit, though there can also be other changes helping. And as shown in Figure 5.7 our mipmap solution shows a slower growth in running time.

Unfortunately, none of the recorded speeds are near what are needed for real-time graphics. But based on the speed increase shown when moving from the GTX 260 to the GTX 480 we believe there is still hope for the technique. It will require optimising the matching routine and possibly an alternative sampling strategy, but we do think an algorithm built around using mipmaps could be a viable solution.

6.3.5 Future work

To make it more viable there are several possible avenues to explore.

Whilst we are making some use of vector types and operations, it could be possible to make use of these to a greater extent with the potential speed increases that would bring.

Then there is the whole integer versus floating point. Whilst the GTX 480 is fast, Voicu [16] have shown that in the fragment shader integer operations are still done at half speed compared to floating point operations, so it would be interesting to change over to using floating points and see how much and effect that will have on the speed.

It could also be worthwhile to explore alternative methods of doing the modulo operation. Bit-wise operations are a new addition to GPU programming, and so potentially not as optimised as other operations, but doing bit-wise comparison could prove to be a speedier alternative.

Another area that could have a large impact on the speed of the algorithm is the sampling strategy used. We outlined some possible strategies Section 6.3 on taking one sample, multiple samples or a combination of the two that would be natural to test.

In addition we would like to test how much of a speed increase could be gotten by sampling the entire 1-dimensional strips, caused by odd coordinates, in one pass. This could have a drastic effect by only running the matching algorithms once for the odd areas instead of multiple times as now. In addition it could help reduce the number of variables on the stack.

For those GPU architectures that support it, it would also be natural to see how they perform when using a dynamic stack, like a linked list, compared to our fixed size stack.

6.4 SAT solution

In keeping with the summed mipmap pyramid idea, though changing the underlying data structure, we decided on using an approach based on using summed area tables (SAT). It has been in use in other areas of computing, but, as far as we know, never before for this problem.

Where the mipmap pyramid have several levels, and the positions at the higher levels corresponding to a larger area covered, the SAT consist of a single level, though the value at a given position is the sum of itself and all previous values below and to the left of itself.

To sample an arbitrarily large rectangular area, within the confines of the map, simply take a sample at each corner of the rectangle, and subtract and add them together as seen in Figure 3.6.

This means that we are able to sample the interesting radii by taking 4 sample and without the complex matching algorithm we needed for our mipmap solution.

Similarly to the mipmap solution the SAT solution is not bound to a specific search routine but are free to choose the routine best suited for the scene.

6.4.1 Implementation

Now let's take a quick look at our SAT implementation. The full details can be found in Section 4.2.

Like our mipmap implementation, our SAT implementation is split into two parts. The first part is to generate the SAT a task we based on the article of Hensley et al. [6]. For details on how it is done, see Section 3.4.1. We implemented the generating routine twice, once taking 2 samples per pass, and one taking 4 samples per pass, which we figured would be sufficient to test our solution.

With the SAT built, we implemented both a forward search and a binary search to find the radii. The sampling function simply takes the 4 samples needed, with conditionals to handle cases where the area goes outside the density map, calculates the value found. This is then scaled by any area outside the density map, and the result returned to the search function.

The search function then determines if the search should continue or terminate and store the radii.

6.4.2 Pros

The most obvious advantage of using a SAT for the search is the need to only take 4 samples to cover an arbitrarily large rectangular area. This is a dramatic reduction compared with not only the brute force, but also our mipmap solution.

Furthermore, unlike our mipmap solution, it has a very low complexity in the code, since there is no costly matching needed to find where to take the samples.

And by being a very light weight solution, according to AMDs GPU Shader Analyzer it only uses 10 general purpose registers, it should be cheap to swap in and out allowing the GPU scheduler to keep many shading quads in flight at the same time.

Lastly, as the sampling function is not bound to any search strategies, one can freely choose the one most suited for the most likely photon distribution in the scene.

6.4.3 Cons

One potential problem with this solution is the need for conditional reads on the four samples to handle the cases where the sample area goes outside the density map. This means that fragments in the shading quad might have to wait on each other when one or more of them samples a region crossing the texture border.

There is also the issue with coalescent reads and texture cache. Depending on the search routine used, like when using a binary search, the divergence in what radii to search makes it unlikely that the texture fetches can be done with coalescent reads, but have to be treated as separate reads. For the same reason the chances of finding the values to be sampled in the texture cache is unlikely as n grows larger.

It also is the solution requiring the most memory on the graphics card, needing $2n$ of texture space for the SAT if the input needs to be preserved. But this is unlikely to a problem in most cases.

6.4.4 Summary

As we have seen in Section 5.5 our SAT solution is quite fast and able to find fill the radius map for a problem size of 512 in only 1.587ms when run on the GTX 260 and taking only 2 samples per pass. Taking 4 samples the time decreases to only 1.357 ms. This then speeds up even further when the faster GTX 480 is used.

When used with a binary search the time it takes to complete the search is dependant on the maximum radius. Figure 5.13 shows how the solution performs for different mr. The values shown are more or less constant, though they can be affected by other operations using the GPU, and so makes it easy to calculate if the cost of the search is acceptable.

We had expected the SAT to suffer from uncoalesced memory access when searching, but it is hard say if this is a problem since the search speed is as fast as seen in Figure 5.11.

Also it seems unlikely that each the fragments have to wait much on each other, despite the conditionals to avoid sampling outside the edges, see Section 4.2. Any divergence here will probably be covered by the GPU swapping between shading quads to hide the texture fetches latency anyway.

The cost of generating the SAT is shown in Figure 5.12, and whilst this is a relatively expensive step in itself it is not prohibitive. And it would not be surprising if it was possible to optimise this step further.

6.4.5 Future work

As we noted the current SAT generation is reasonably quick, though it would be interesting to see if this could be optimised further. This could for instance be through taking even more samples per step or by optimising the steps so that only those values not yet completed are worked on.

It could also be interesting to check if one could implement the sampling differently. One possible optimisation could be to fetch all 4 samples at the start of the shader and then calculate whether or not they should affect the final sum. This could be very hardware, driver and compiler specific, but could none the less be an interesting experiment.

It could also be interesting to see if there are ways to partially update the SAT, allowing for photons to be traced into the scene over several frames to increase quality.

6.5 Outside handling

The way we handle the area outside the density map, described in Section 3.1 and section 4.4, has some effect on the speed of the algorithms, though as it is a purely mathematical approach without conditionals it should be minimal. And ignoring the outside area, however tempting that may be, will most likely result in radii larger than necessary. We did not have time to go thoroughly into this, but would like to explore this further in future work.

Chapter 7

Conclusions

For this thesis we have focused on how it is possible to find the radii needed to use variable sized filter kernel when implementing photon mapping on the GPU, without the use of *k*d-trees.

In Chapter 3 we have shown two new methods, in addition to a naive brute force approach, on possible ways this can be accomplished.

Taking the results from Chapter 5 into account, we can see that our solution based around a summed area table (SAT), see Section 3.4 and 6.4, is fast enough to be implemented into existing graphics pipelines without incurring any large performance penalties. Whilst it does have some size requirements, as well as requiring a few passes to set up the SAT itself, we believe that these are well with reasonable limits.

Our solution based around a summed mipmap pyramid, see Sections 3.3.2 and 6.3, does not perform as well. Though it allows for interactive frame rates, it is a far cry from the speeds achieved by the SAT solution. But as we note in Section 6.3.4 we do believe that the method has potential, and it would be interesting to see how it performs with the suggestions from Section 6.3.5.

Even the naive brute force solution performs rather well, and with the optimizations mentioned in Section 6.2 it could also become a usable technique, especially when memory is constrained.

As noted in Chapter 5 we feel it would have been interesting to test with other data sets with differing photon density, as well as test with the existing data set though vary the number of photons required. However, due to time constraints this was not possible, but is something that could be explored in the future.

Chapter 8

Acknowledgements

In the writing of this master thesis there are several people that have been a great help. First and foremost our supervisors E. Christopher Dyken and Martin Reimers who have provided invaluable help, hint, support and encouragement, as well as giving feedback on structure, grammar as well as outright holes. All remaining flaws are solely there by our own doing.

Further we would like to thank PhD students André R. Brodtkorb and Martin L. Sætra as well as fellow Master student Martin Ertsås for the insightful discussions, the rubber ducking and the encouragements.

In addition we would like to thank Sintef ICT for providing both an office and the hardware used for development and testing.

Finally we would like to thank our family for the support, encouragement and backing we have received when writing this thesis. Without them it probably would not have happened.

Appendix A

Tables

In this appendix we have the tables containing the data used to build the graphs in Chapter 5.

Mipmap generation only		
Size	**GTX 260**	**GTX 480**
64	0.342	0.340
128	0.344	0.341
256	0.348	0.346
512	0.362	0.344
1024	0.418	0.342

Table A.1: Time in ms to manually generate summed pyramid mipmaps from $Size^2$ to 1.

Mipmap generation and search				
Size	**GTX 260 Forward**	**GTX 260 Binary**	**GTX 480 Forward**	**GTX 480 Binary**
64	106.913	104.817	30.937	30.937
128	162.563	201.567	52.690	62.280
256	422.467	586.667	113.367	182.960
512	1346.133	1914.800	315.667	476.467

Table A.2: Time in ms to manually generate and search mipmaps.

GTX 260 SAT generation and search				
Size	**SAT 2 Forward**	**SAT 2 Binary**	**SAT 4 Forward**	**SAT 4 Binary**
64	0.247	0.248	0.165	0.164
128	0.274	0.275	0.193	0.190
256	0.384	0.464	0.297	0.382
512	1.233	1.587	0.999	1.357
1024	8.053	5.498	7.049	4.480

Table A.3: Time in ms to generate and search SATs taking 2 and 4 samples using GTX 260.

GTX 480 SAT generation and search				
Size	**SAT 2 Forward**	**SAT 2 Binary**	**SAT 4 Forward**	**SAT 4 Binary**
64	0.232	0.233	0.157	0.154
128	0.259	0.259	0.181	0.180
256	0.305	0.290	0.245	0.227
512	0.843	0.823	0.762	0.743
1024	4.354	2.952	4.016	2.614

Table A.4: Time in ms to generate and search SATs taking 2 and 4 samples using GTX 480.

SAT generation only				
Size	**GTX 260 SAT 2**	**GTX 260 SAT 4**	**GTX 480 SAT 2**	**GTX 480 SAT 4**
64	0.222	0.145	0.214	0.136
128	0.249	0.169	0.2474	0.162
256	0.277	0.172	0.270	0.165
512	0.680	0.449	0.546	0.465
1024	2.690	1.687	1.885	2.062

Table A.5: Time in ms to manually generate SATs taking 2 and 4 samples per pass.

SAT search only

Size	GTX 260 Forward	GTX 260 Binary	GTX 480 Forward	GTX 480 Binary
64	0.040	0.045	0.034	0.015
128	0.059	0.081	0.040	0.024
256	0.164	0.247	0.099	0.080
512	0.550	0.909	0.294	0.275
1024	5.365	2.796	2.464	1.064

Table A.6: Time in ms to search the SAT with forward and binary search.

SAT binary search, density map size = 512

Max radius	GTX 260	GTX 480
32	0.909	0.275
64	1.023	0.326
128	1.137	0.377
256	1.249	0.427

Table A.7: Table showing how the search time varies for binary searches based on the max radius.

Brute force search

Size	GTX 260	GTX 480
64	17.987	15.300
128	47.830	23.777
256	187.287	96.660
512	687.600	392.200
1024	2900.800	1514.013

Table A.8: Table showing how the search time varies for binary searches based on the max radius.

Bibliography

[1] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[2] Franklin C. Crow. Summed-area tables for texture mapping. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 207–212, New York, NY, USA, 1984. ACM.

[3] E. Christopher Dyken. http://folk.uio.no/erikd/.

[4] Wolfgang Engel. *ShaderX7*, chapter Designing a Renderer for Multiple Lights - The Light Pre-Pass Renderer, pages 467–478. 2009.

[5] Fernando and Randima, editors. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter 37.3.1 Bitonic Merge Sort, pages 627–630. Pearson Higher Education, 2004.

[6] Justin Hensley, Thorsten Scheuermann, Greg Coombe, Montek Singh, and Anselmo Lastra. Fast summed-area table generation and its applications. *Computer Graphics Forum*, 24:547–555, 2005.

[7] Robert Herzog, Vlastimil Havran, Shinichi Kinuwaki, Karol Myszkowski, and Hans-Peter Seidel. Global illumination using photon ray splatting. In Daniel Cohen-Or and Pavel Slavik, editors, *Computer Graphics Forum (Proceedings of Eurographics)*, volume 26(3), pages 503–513, Prague, Czech Republic, 2007. Blackwell.

[8] David Luebke. Surveying real-time beyond programmable shading rendering algorithms. Published online. Beyond Programmable Shading, Course Siggraph 2010 http://bps10.idav.ucdavis.edu/.

[9] Morgan McGuire and David Luebke. Hardware-accelerated global illumination by image space photon mapping. In *Proceedings of the 2009 ACM SIGGRAPH/EuroGraphics conference on High Performance Graphics*, New York, NY, USA, August 2009. ACM.

[10] Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. Optix: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA, 2010. ACM.

[11] Christophe Pham. Photon mapping. Published online. http://www.virtual-eyes.net/fr/projecs/52-photon-mapping.html.

[12] Aras Pranckevičius. Compact normal storage for small g-buffers. web, march 2010. http://aras-p.info/texts/CompactNormalStorage.html.

[13] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics*, 21(3):703–712, July 2002. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

[14] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50. Eurographics Association, 2003.

[15] Wolfgang Stürzlinger and Rui Bastos. Interactive rendering of globally illuminated glossy scenes. In *Proceedings of the Eurographics Workshop on Rendering Techniques '97*, pages 93–102, London, UK, 1997. Springer-Verlag.

[16] Alex Voicu. Nvidia fermi gpu and architecture analysis. Published online. http://www.beyond3d.com/content/reviews/55.