Hardware Acceleration of Photon Mapping

Carl Andrew Hoggins

A thesis submitted for the degree of Doctor of Philosophy (Ph.D) at Newcastle University

School of Electrical, Electronic and Computer Engineering

August 2010

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express my gratitude to my Ph.D supervisor, Dr. J. N. Coleman. He has continuously stimulated my research interests, provided valuable constructive criticism and been a constant source of encouragement.

I would also like to thank my family and friends, as without their support finishing this thesis would not have been possible.

SUMMARY

The quest for realism in computer-generated graphics has yielded a range of algorithmic techniques, the most advanced of which are capable of rendering images at close to photorealistic quality. Due to the realism available, it is now commonplace that computer graphics are used in the creation of movie sequences, architectural renderings, medical imagery and product visualisations.

This work concentrates on the photon mapping algorithm [1, 2], a physically based global illumination rendering algorithm. Photon mapping excels in producing highly realistic, physically accurate images.

A drawback to photon mapping however is its rendering times, which can be significantly longer than other, albeit less realistic, algorithms. Not surprisingly, this increase in execution time is associated with a high computational cost. This computation is usually performed using the general purpose central processing unit (CPU) of a personal computer (PC), with the algorithm implemented as a software routine. Other options available for processing these algorithms include desktop PC graphics processing units (GPUs) and custom designed acceleration hardware devices.

GPUs tend to be efficient when dealing with less realistic rendering solutions such as rasterisation, however with their recent drive towards increased programmability they can also be used to process more realistic algorithms. A drawback to the use of GPUs is that these algorithms often have to be reworked to make optimal use of the limited resources available.

There are very few custom hardware devices available for acceleration of the photon mapping algorithm. Ray-tracing is the predecessor to photon mapping, and although not capable of producing the same physical accuracy and therefore realism, there are similarities between the algorithms. There have been several hardware prototypes, and at least one commercial offering, created with the goal of accelerating ray-trace rendering [3]. However, properties making many of these proposals suitable for the acceleration of ray-tracing are not shared by photon mapping. There are even fewer proposals for acceleration of the additional functions found only in photon mapping.

All of these approaches to algorithm acceleration offer limited scalability. GPUs are inherently difficult to scale, while many of the custom hardware devices available thus far make use of large processing elements and complex acceleration data structures.

In this work we make use of three novel approaches in the design of highly scalable specialised hardware structures for the acceleration of the photon mapping algorithm. Increased scalability is gained through:

- The use of a brute-force approach in place of the commonly used smart approach, thus eliminating much data pre-processing, complex data structures and large processing units often required.

- The use of Logarithmic Number System (LNS) arithmetic computation, which facilitates a reduction in processing area requirement.

- A novel redesign of the photon inclusion test, used within the photon search method of the photon mapping algorithm. This allows an intelligent memory structure to be used for the search.

The design uses two hardware structures, both of which accelerate one core rendering function. Renderings produced using field programmable gate array (FPGA) based prototypes are presented, along with details of 90nm synthesised versions of the designs which show that close to an order-of-magnitude speedup over a software implementation is possible. Due to the scalable nature of the design, it is likely that any advantage can be maintained in the face of improving processor speeds.

Significantly, due to the brute-force approach adopted, it is possible to eliminate an often-used software acceleration method. This means that the device can interface almost directly to a front-end modelling package, minimising much of the pre-processing required by most other proposals.

LIST OF ABBREVIATIONS

2D	Two Dimensional
3D	Three Dimensional
AABB	Axis-Aligned Bounding Box
A*k*NN	Approximate *k* Nearest Neighbours
ALU	Arithmetic Logic Unit
ASIC	Application Specific Integrated Circuit
B-*k*D	Bounded *k* Dimensional
BRDF	Bidirectional Reflectance Distribution Function
BSP	Binary Space Partitioning
BV	Bounding Volume
BVH	Bounding Volume Hierarchy
CAM	Content Addressable Memory
CMOS	Complementary Metal Oxide Semiconductor
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
ELM	European Logarithmic Microprocessor
FLP	Floating-Point
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphics Processing Unit
IC	Integrated Circuit
*k*D	*k* Dimensional
*k*NN	*k* Nearest Neighbours
LNS	Logarithmic Number System
LSH	Locality Sensitive Hashing
MBR	Minimum Bounding Rectangle
PC	Personal Computer
PS	Photon Search
RAM	Random Access Memory
RAS	Ray Triangle Intersection Acceleration Structure
RLS	Recursive Least Squares
ROM	Read Only Memory
RTI	Ray Triangle Intersection
SAH	Surface Area Heuristic
SIMD	Single Instruction Multiple Data
SPU	Shader Processing Unit
SRAM	Static Random Access Memory
VLSI	Very Large Scale Integration
εN	Euclidian Neighbours

1 INTRODUCTION

Computer generated graphics have become very widely used in recent years. It is now commonplace for product visualisations, movie sequences and medical imagery to make widespread use of such graphics. The quest for realism in images has yielded a range of computer graphics algorithms of increasing complexity.

The first renderers, or computer graphics programs that synthesise an image, were primitive based. Primitives are simple geometric objects that are used to describe the scene to render in a three-dimensional (3D) model, the most common of which is the triangle. Such rendering techniques were very successful and in fact one primitive based renderer, rasterisation, is still used in most commodity graphics processing hardware. The major drawback to this class of rendering is the lack of realism in the images generated. While it is possible to add effects such as shadows and reflections, these are often only textures which do little to enhance realism, although are often used to make renderings more visually acceptable.

The next class of renderers were pixel based, which offer enhanced realism in the images generated over their primitive based predecessors. These operate on a pixel by pixel basis, calculating what primitive within a model will be visible at each pixel. Ray based renderers, which were the first to take any significant account of the physical behaviour of light through accurate representation of reflections and refractions, fall within this category.

Ray-tracing is the most prominent and successful ray based renderer. To determine which model primitives are visible in the image being generated using ray-tracing, multiple rays are cast from the viewer's eye, through a pixel in a representation of the image being created, and into the geometric description of the model. At the point where the ray intersects a model primitive, the colour of the primitive will contribute to the colour of the pixel. In the case of reflective or transparent surfaces, any reflected or refracted ray paths will be followed and where they too intersect a primitive its properties will be added to that of the primary intersection point. This means that ray tracers produce reasonable results in areas of specular, or focused, reflection from shiny surfaces, but do not account for the unfocused scattering from matte ones which generally accounts for most of the background illumination. Instead, an ambient lighting coefficient is often used.

The radiosity algorithm is a common addition to ray tracing, its objective being the calculation of indirect, or background, illumination in a scene with increased realism. The algorithm calculates the aspect of every matte surface in a scene relative to each of the others, and evaluates the total reflection of background light around the scene. This technique is not appropriate for specular reflection, and it is therefore common to let radiosity handle indirect illumination and ray tracing the direct illumination to account for all possible surface types in the model.

One of the most realistic rendering algorithms available is photon mapping. This addresses the deficiencies of radiosity as the artificial distinction between specular and matte surfaces is eliminated. Photon mapping performs a much more realistic evaluation of the physics of light. Initially multiple photons, or specialised rays carrying light energy, are emitted from a light source into the scene. The surface qualities are evaluated at any point where a photon intersects a primitive. Reflection and transmission do not necessarily have to take place in one direction only; a matte surface can reflect fractions of the incoming energy in many different directions. One or more secondary photons carrying a proportion of the energy of the original will then be emitted from the intersection point, and these too are intersected with the model. A 'map' is built up showing the intersection points of all photons that have arrived at reflective surfaces, as reflected light is visible to the viewer. After this initial map building stage, the image can be generated. Image generation is much the same as in a ray-tracer, however there is an extra step in locating all photons stored in the vicinity of every ray-model intersection point. The power of these photons is averaged to arrive at an estimate of the illumination at that point.

Photon mapping is able to rectify many of the realism issues of earlier algorithms, however by doing so other problems have developed. These problems are centred on algorithm execution time, and therefore the time needed to generate these realistic images. The most commonly used functions within a photon map renderer are those which determine which model primitive a ray intersects and searching the photon ‘map’ for photons around these intersection points. These functions contain large amounts of reasonably complex computation, which is the driver behind photon mapping having longer execution times than most other renderers.

In order to reduce the time taken to generate photon mapped images, several software acceleration techniques have been developed. For the ray-primitive intersection testing, these techniques are concerned with reducing the number of primitives that must be tested with each ray and work by subdividing model geometry or spatial areas of the model into volumes. These volumes can then be tested for ray intersection. If a ray does not intersect the volume, it cannot intersect any primitive contained within this volume. A common acceleration technique for photon searching works by caching previously computed illumination values, which are reused via interpolation. This reduces the total number of photon searches that must be performed.

In terms of hardware acceleration, we are not aware of any capable of accelerating the photon mapping algorithm in its entirety. However, there have been prototypes and also commercial offerings of acceleration devices for the ray-primitive intersection functions in the context of ray tracing applications. Many of these designs are not ideally suited for use in photon mapping as they depend on processing multiple similar rays concurrently or having a pool of rays pre-initialised in memory. Rays generated in photon mapping often exhibit increased randomness over those in ray-tracing and are serially dependant, making this difficult. It is also clear that many of these proposals do not lend themselves to large-scale parallelisation, due to their size and use of complex data structures. There have also been proposals for GPU-based solutions, where parallelisation would be inherently difficult due to the limited processing resources available and the difficulty in spreading computation across multiple devices. Like their software equivalents, many of these proposals depend on spatial or geometry subdivision of the model to reduce the number of intersections to perform. There are very few proposals for the acceleration of photon searching using hardware. Those that do exist either attempt to modify the algorithm so that it becomes easier to execute using a GPU, which in some cases reduces accuracy, or use small-scale parallelisation of tree traversal units in custom hardware. Again, both of these classes of proposals lack scalability which limits the acceleration that can be achieved.

The aim of this work is the acceleration of the photon mapping in its entirety using custom hardware acceleration devices. To achieve this objective, we must first determine in which rendering functions the majority of execution time is consumed. We perform an analysis of our own software photon map renderer to this end. A profiling routine within the software revealed that three functions dominate total execution time. Ray-triangle intersection (RTI) processing determines the intersection points of all the rays (including photons) fired around the scene, with the primitives (which are triangles in this case) that form the scene. The processing of an RTI acceleration structure (RAS), which spatially subdivides the scene to reduce the number of RTIs to compute is also time consuming. Finally the photon search (PS) function is used to find all the photons in the vicinity of an intersection point, and is shown to add significantly to overall execution time.

We describe a proposal for hardware units to accelerate processing of the RTI and PS functions. These hardware acceleration units have three aspects of novelty:

1. Both devices are based on a brute-force approach and are highly scalable.
2. This scalability is facilitated by the reduction in processing hardware area through the use of LNS arithmetic elements.
3. The novel PS acceleration unit design is based on a highly parallel intelligent memory unit.

The RTI acceleration device is capable of operating in this brute-force mode, scaled to the extent that significant acceleration of the algorithm as a whole can be gained without employing any RAS technique, and also a smart manner which does make use of a RAS. A significant advantage of the former approach being the elimination of any pre-processing associated with RAS generation. In this case the hardware need only be configured with simple triangle data such as that output almost directly from any modelling utility. Details of the processing performed by each device, their structure and operation are presented.

The two units have been specified and designed in a hardware-description language. They have been prototyped and extensively tested on FPGA devices. The designs have also been synthesised in 90nm technology. We show renderings produced using FPGA prototypes of each device, and present silicon area requirements for ICs containing varying numbers of each processing element. We use the operational speeds which were also identified during IC synthesis to show that, using parallel arrays of each device, an order-of-magnitude reduction in execution time over the equivalent software functions is achievable in the rendering of typical models, leading to close to order of magnitude reductions for the algorithm as a whole. We also discuss the usage of the acceleration units and their interface to a host running the remaining photon mapping functions.

2 BACKGROUND OF COMPUTER GRAPHICS

The main aim of this work is the acceleration of one of the most advanced and realistic rendering algorithms in photon mapping. In this section we give a brief overview of computer based rendering to this point before moving on to look at photon mapping in more detail.

2.1 3D Models

A model contains a representation of a scene to be rendered. These scenes on which an image is based are composed of geometric objects of varying shape and size which are oriented in 3D space.

To reduce the number of functions within a renderer which deal with the multitude of possible geometric shapes, it is common to decompose these into a single type, or primitive. The triangle and polygon are common choices, due to their simplicity and their ability to accurately represent even the most complex geometric shapes. All models used in this work are composed solely of triangles.

2.2 Renderers

As stated previously, a renderer is a computer program that synthesises an image of a model supplied. There are two main classes of renderer; primitive based and pixel based. Primitive based renderers geometrically project objects in the model to an image plane without advanced optical effects. The second class of renderer is pixel based, in which all pixels in an image must be processed, no matter whether they show objects or not.

2.2.1 Primitive Based Renderers

In this class of renderer, model geometry is projected onto an image plane. Every model primitive is investigated to determine which pixels, if any, these primitives affect, with the pixels then being modified accordingly. As there is little computation to be done for large parts of the image where no objects appear, this process can be performed very quickly. A drawback to such algorithms is the realism achievable, which is often lacking.

Rasterisation is the most common primitive based renderer. In this algorithm, a scene is described using collections of triangles in 3D space. The three vertices of these triangles are simply transformed from 3D space into corresponding two dimensional (2D) points in an image, with the triangles formed being in-filled as appropriate.

As rasterisation is simply the process of mapping the model geometry to pixels, there is no particular way to compute the pixel colours. Shading algorithms, such as Gauraud and Phong [4, 5], are often used to modify pixel colours giving a greater effect of realism. These shading routines allow objects to exhibit a greater degree of realism through more realistic representation of curved surfaces and specular, or focused, highlights.

As the required transformations are simple, these can be performed very efficiently and quickly in hardware. For this reason, rasterisation based graphics pipelines have been used in most commodity graphics hardware of the recent past, even though current graphics hardware often contains many other resources and is increasingly programmable [6].

Another primitive based renderer is scanline rendering. This operates on a row-by-row basis, where all polygons within a model are sorted and stored in a list according to the top *Y* coordinate at which they first appear in an image. Each row of pixels within the image, or scan line, is computed using the intersection of this scan line with the polygons on the front of the sorted list. The sorted list is then updated to discard polygons which can no longer be visible as the active scan line is advanced.

In order to realise depth perception, hidden surface removal algorithms [7, 8] such as *z*-buffering can be used to determine which surfaces are 'behind' others from the viewer's perspective. A *z*-buffer, which is a two dimensional array with one array element for each image pixel, is used to store the depth (*z*-coordinate) of the primitive shown in any pixel. When rendering any other primitive, the *z*-buffer elements of the pixel locations of where this primitive would appear in the rendered image are examined. If the depths of the new primitive are less than the values already stored in the *z*-buffer, the primitive is closer to the viewer and the pixels are updated.

2.2.2 Pixel Based Renderers

Pixel based renderers are often implemented as ray-based routines, where rays are used to represent the light travelling between the viewer and model, through each pixel in an image. Rays are cast from the viewer's eye position, through a 2D grid representing the pixels in the rendered image, towards the model. These rays are then tested for intersection with all primitives forming the model. The properties of the first intersected primitive along the ray contribute towards the colour of the corresponding pixel.

Processing each of the rays representing light transfer, at least one for each pixel in an image, is computationally intensive and as a result renderers in this class often have longer run times than their primitive based counterparts. However this drawback is often outweighed by the increase in realism gained.

2.2.3 Ray Tracing

The most prominent ray based renderer is ray-tracing [9-11]. Ray-tracing operates by casting rays from the viewer's position towards the model being rendered. If any model geometry is intersected by a ray, the surface closest to the viewer is selected. The illumination at the point where the ray intersects this surface is calculated.

To calculate the illumination, the distance to the scene light sources can be determined, or a less realistic ambient lighting term can be used. To see if any intersection point is in shadow, a shadow ray is fired from this point towards any light sources. The intersection point is in shadow from a light source if the shadow ray intersects any model geometry before reaching this source.

In addition to shadow rays, further rays are fired depending on the properties of the surface intersected. If a surface is reflective, the direction of the reflected ray is calculated and a ray is emitted in this direction from the intersection point. This is also the case with transmissive surfaces. It is common to allow several levels of recursive rays to be fired for reflections and refractions. The illumination and colour at each recursive level of ray contributes to the previous level, meaning these rays also contribute to the illumination and colour of the previous surface intersected, and thus also in some part the final pixel colour.

This form of ray-tracing is often referred to as 'backward' ray-tracing, as rays are fired in the opposite direction to the rays of light represented. Therefore, the intensity of the rays at the maximum recursive level (the most reflected or refracted) are calculated first, with these contributing to the intensity of rays as the level of recursion reduces depending on the particular properties of the surfaces intersected. Finally, the intensity of the primary viewer rays (at the minimum recursive level) are calculated and used as the pixel colours in the image created.

An advantage of ray tracing is its algorithmic simplicity, however there is a drawback to this simplicity as only a basic approximation to the rendering equation [12] is used. The rendering equation is based on the law of conservation of energy, and states that outgoing light is the sum of all reflected and emitted light, and that reflected light is based on incoming light from all directions, the angle at which this light arrives at a surface and also the properties of the surface itself. This means that renderings generated using ray-tracing will never exhibit advanced optical effects such as caustics (the focusing of light on a surface), or colour bleeding (the colour of one surface bleeding onto another nearby) without the incorporation of further techniques.

While it is not possible to produce photorealistic images using ray tracing, visually pleasing results are achievable. One factor in this is the ability of ray-tracing to accurately deal with specularly reflective surfaces, and other optical effects such as refraction. Another is the ability to handle the creation of shadows better than primitive rendering algorithms, although these often have inaccurate sharp edges.

However, the most prominent drawback to ray-tracing is rendering times, which can be significantly longer than other, albeit more basic, techniques. This is due to the large amounts of complex calculation which must be performed, mainly in determining which model primitives the many rays intersect. It follows that the execution time of ray-tracing is critically dependent on the time it takes to perform these calculations.

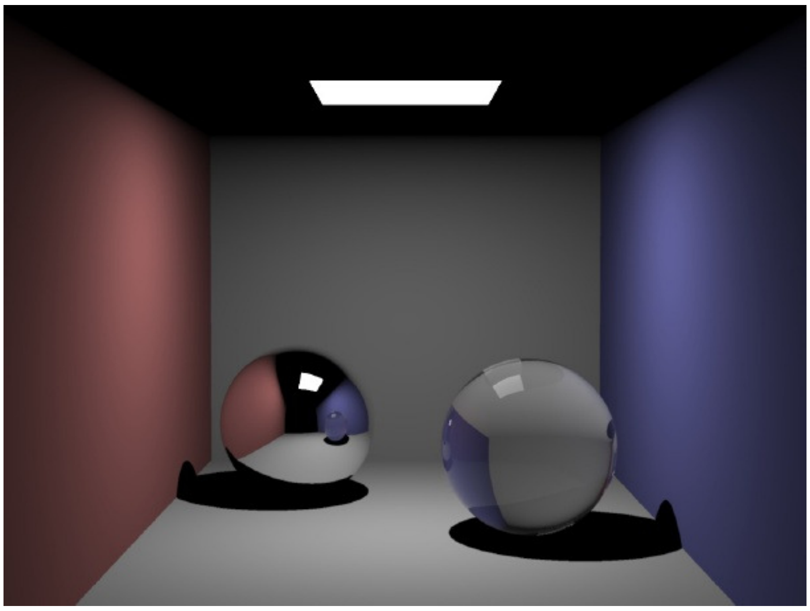


Figure 1: Ray traced Cornell Box scene [13]. Notice the realistic reflections and refractions, but less realistic illumination and hard shadowing.

Radiosity and photon mapping are fairly recent improvements to the ray-tracing algorithm, which transform the renderer into a global illumination solution, capable of both direct and indirect, or background, illumination. These additions improve accuracy of the illumination, and therefore the image as a whole, by modelling the transfer of light energy throughout a scene more accurately.

2.2.4 Radiosity

Radiosity [14] was first used in the field of heat transfer, later being refined to deal with the transfer of light energy throughout a model.

Radiosity is not a rendering algorithm in its own right, but rather an extension to algorithms such as ray-tracing. While ray-tracers are able to produce reasonable results for areas of direct illumination and focused reflection or refraction from shiny surfaces, no account is taken of the unfocused scattering of light energy between matte surfaces in a scene. It is this scattering between the diffuse surfaces that generally accounts for much of the background illumination.

Radiosity only deals with the transfer of energy between the diffuse (Lambertian) surfaces in a model. Diffuse surface patches, each with a defined initial energy value (or illumination) are used. The energy of each patch is spread around all other diffuse patches depending on the relationship (or form factor) between each pair of patches, which is based on distance and patch orientation. A combined procedure implementing both ray-tracing and radiosity means that illumination containing both direct and indirect components can be realised in an image generated.

In the use of radiosity, difficulties can arise in the trade-off between patch size and the number of discrete surface patches. With more surface patches, the indirect illumination calculations will take longer, however with too few patches the accuracy of the background illumination will be impacted.

2.2.5 Photon Mapping

Much like radiosity, photon mapping [2] can be considered an extension of the ray-tracing algorithm, which improves the accuracy of the illumination calculation at any point on an intersected surface. Photon mapping eliminates the artificial distinction between specular and matte surfaces, performing a much more realistic evaluation of the physics of light which results in a higher degree of image realism.

To show the difference in features and the effect they have on the images generated, graphics from a SIGGRAPH course [13] are shown in Figure 1 where the model has been ray-traced, and Figure 2 where photon mapping was used. These images are rendered using the same model, a slightly modified Cornell Box scene.

The photon mapping algorithm has two stages. In the first stage photons are emitted from light sources into the scene. When these photons intersect scene geometry they are not necessarily reflected or refracted in one direction only; a matte surface will reflect fractions of the incoming energy in a multiplicity of directions. One or more secondary photons carrying a proportion of the energy of the original will then be emitted from the intersection point, and these too are intersected with the model. A 'map' is built up containing the intersection point and intensity of all photons that have arrived at reflective surfaces and which are therefore visible to the viewer. Unlike rendering rays, these photon rays are ‘forward rays’ as they travel with the direction of light.

In the second stage backward rays are used, similar to a ray tracer, where it is determined which point in the model is visible at each pixel. Recursive rays are again used to deal with reflections and transmissions. Calculating the illumination at each ray-model intersection point involves finding all stored photons within a pre-set radius of this point and averaging their intensities over the area of the intersected surface bounded by the volume searched. This photon search radius has to be set such that sufficient photons will have been stored within it to give a statistically accurate estimate of the illumination at that point.

Practical rendering requires a balance between the search volume and the number of photons emitted. A larger search volume permits an image to be rendered with fewer photons emitted, but allows photons to have an effect on points remote from where they themselves were reflected from the scene. Their energy is effectively smeared over a larger area, blurring areas of fine detail in the lighting. Conversely, using a smaller search volume increases the likelihood that a particular point will not be within range of a statistically representative total of photons, resulting in an incorrectly low estimate of the illumination there and a noisy image.

The most accurate approach is to fire a large number of photons and use a small radius, although the preferred approach to computational efficiency is to do the opposite, often in conjunction with an adaptive method that locally adjusts the search area in a region of rapidly changing illumination. This is particularly important in the rendering of caustics, the focusing of light by reflective or transmissive materials. The most common adaptive method is the *k* nearest neighbours (*k*NN) search, where the *k* photons closest to the query point are used, with the distance to the furthest one in this set used in the density calculation.

Advantages of photon mapping over ray tracing include the ability to generate markedly more realistic images, as photon mapping makes use of global illumination, resulting in phenomena such as colour bleeding being visible in rendered images. Realistic shadows are also created automatically negating the need for workarounds such as shadow rays, although shadow photons can be used to reduce the overall number of photons to be fired [2].

Although there are several advantages of using photon mapping over ray tracing, there is one major drawback. This is the time taken to produce images, which is often significantly longer than the equivalent ray trace render time. This is again due to the large amounts of complex calculation which must be performed, much in determining which model primitives the many rays intersect but additionally in performing the many photon searches required for illumination calculations. It is on these functions that photon mapping render time is critically dependent. In section 3 we present a timing analysis of our photon map renderer for a typical model, showing the breakdown of time spent in these functions.

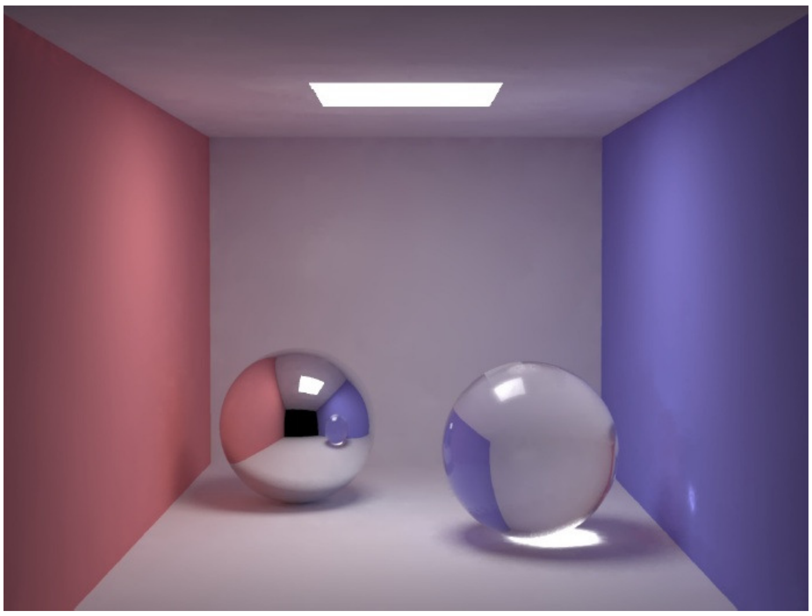


Figure 2: Photon mapped Cornell box scene [13]. Notice the realistic reflections and refractions, caustics, soft edged shadows and realistic illumination especially on the room ceiling.

In order to gain an understanding of how the main rendering functions of photon mapping coexist, pseudo code for a basic photon map renderer is presented in Figure 3.

```
public Colour ray_colour;
void RenderModel (Model m){
       PhotonMap pm = GeneratePhotonMap(m);
       GenerateImage(m, pm);
}
PhotonMap GeneratePhotonMap(Model m){
       PhotonMap pm;
       Coordinates c;
       PhotonIntensity it = lightIntensity;
       For i = 0 to Num_Photons do
              c = LightOrigin;
              FirePhoton(m, pm, c, it);
       Next i
       Return pm;
}
void GenerateImage(Model m, PhotonMap pm){
       Ray r;
       Coordinates c;
       For i = 0 to Num_Rays do
              c = ViewOrigin;
              ray_colour = BLACK;
              FireViewerRay(m, pm, c)
              WritePixelColour(ray_colour);
       Next i
}
void FirePhoton(Model m, PhotonMap pm, Coordinates c,
       PhotonIntensity it){
       Ray r = GeneratePhoton(c);
       c = FindClostestIntersection(r, m);
       StorePhoton(c, pm, it);
       If CURRENT_RECURSION_LEVEL < MAX_RECURSION_LEVEL Then
              it = CalculateNewPhotonIntensity();
              FirePhoton(m,  pm,  c,  it);  //for  reflection,  transmission  or
       End if                              //both
}
void FireViewerRay(Model m, PhotonMap pm, Coordinates c){
       Ray r = GenerateViewerRay(c);
       c = FindClostestIntersection(r, m);
       If CURRENT_RECURSION_LEVEL < MAX_RECURSION_LEVEL Then
              FireViewerRay  (m,  pm,  c);   //for  reflection,  transmission  or
       End if                                //both
       ray_colour = ray_colour + CalculateRayColour(c, pm, RADIUS);
}
Coordinates FindClosestIntersection(Ray r, Model m){
       IntersectedGeormetry ig = CalculateGemetryIntersection(r, m);
       CalculateRTI(r, m, ig);
       //return coordinates of first intersection, if any
}
void StorePhoton(Coordinates c, PhotonMap pm, PhotonIntensity i){
       //Store photon with intensity i and coordinates c in photon map pm
}
Colour CalculateRayColour(Coordinates c, PhotonMap pm, Double radius){
       PhotonsFound pf = PhotonSearch(c, pm, radius);
       Irradiance r = CalculateIrradiance(pf);
       //return  colour  of  ray,  calculated  using  irradiance,  colour  of
       //surface  intersected  and  existing  colour  of  ray  and  level  of
       //recursion
}
```

Figure 3: Pseudo code for basic photon map renderer.

3 SOFTWARE ANALYSIS

In the previous section we identified the core tasks of a photon map renderer, which deal with finding which of the many model triangles any particular ray intersects, and in performing photon searches in order to compute the illumination at any point on a model surface.

These tasks map to three rendering functions, these are:

- The ray-triangle intersection (RTI) function. This function determines the first triangle intersected by a ray.
- The RTI acceleration structure (RAS) traversal or processing function. This function traverses a ray through a commonly hierarchical data structure in the aim of reducing the number of triangles that must be tested in the RTI function. The RTI acceleration structure which aids in this is commonly built prior to rendering for static scenes, but continually updated for dynamic ones, and commonly works by spatially subdividing model geometry.
- The photon search (PS) function. This function locates all photons local to a ray-surface intersection which enables an accurate illumination estimate to be computed.

To confirm that these functions are in fact the main contributors to overall rendering time, and discover the proportion that each contributes to overall execution time, a software package implementing a photon map renderer has been analysed.

The software used was designed as part of this work. It includes several features over and above a basic photon map renderer. One of the main additions is a profiling function which provides detailed statistics for both the number of executions and time consumed by various rendering functions. All statistics presented were gained from this software, which was executed on a 3.0GHz Pentium 4 HT based PC with 1 GB of random access memory (RAM) under Microsoft Windows XP SP2. Further details of the rendering software, its features, and also the RAS generators which create the input for the renderer can be found in the following section.

Two RAS schemes are considered. We begin in this section by analysing the use of both *k*-dimensional binary trees (*k*D-trees) and axis-aligned ellipsoid based bounding volume hierarchies (BVH) in order to determine the most efficient rendering setup for our test model. The theory behind these and other techniques that spatially subdivide model geometry can be found in section 4.2.

Using the most efficient RAS setup, we then move on to analyse the photon mapping algorithm in more detail, determining the proportions of time spent in the core rendering functions for a typical setup and also how these proportions vary when the following parameters change:

1. Number of photons emitted from scene light sources
2. Photon search radius for photon map search
3. Model complexity

Determining the time spent in each of the core functions, and taking into consideration how this varies with a change in rendering parameters, will allow us to make an informed decision on whether any particular function should be considered for hardware acceleration.

3.1 Rendering Software

A software photon map rendering suite has been created as part of this work. This is composed of both acceleration structure generators and the photon mapping rendering software.

The photon map rendering software includes several features which improve the physical accuracy of the rendering and therefore increase the realism of the images synthesised. Details of these features, and the methods and techniques on which the rendering software is based can be found in Table 1.

This software has been used in all analysis presented.

3.1.1 kD-Tree Builder

The *k*D-tree generator designed and built as part of this work is based on an outline presented by Havran [15]. Construction is based on a cost model, which calculates the cost of traversing an arbitrary ray through the tree. The cost of traversing each node is based on a surface area heuristic (SAH) [16], which uses the surface area of the node and the number of primitives it contains. For a tree with minimal cost, a node is only split into two child nodes if the sum of the cost of intersecting these nodes plus the cost of traversing another tree level is lower in cost than intersecting the parent node.

The most time consuming task of the generator is in finding the best axis and position at which to subdivide a node. It is common that multiple split position and axis candidates are considered, and the cost of intersecting each of the possible child nodes must be calculated. As computing node costs involves computing surface area and the number of triangles contained, minimising the number of split candidates while minimising tree cost is an active research area.

Our tree builder provides three functions for split candidate identification, each of which varies in generation time and result in trees of varying cost. The three split candidate functions are based on the following:

1. Subdivision of node volume at *x* equally spaced positions in each axis.
2. Subdivision of node volume at the higher and lower bound of each primitive in each axis.
3. Subdivision of node volume at the higher and lower bound of each primitive in each axis, however if there are greater than *x* primitives in a node, the upper and lower bounds of these primitives are found and ordered. Then, *x* positions are used from this ordered set.

The first of these methods has the shortest build times (depending on the number of splits to attempt), however usually results in a costlier tree than the second method, which has significant runtimes but should generate the least costly tree. The third method is a compromise between these two approaches which has proven to produce similar quality trees to the second option with reduced generation time. In terms of tree quality and cost, using the second and third approaches, for some test models, we are able to get close to the three RTI calculations per ray that is considered optimal for a well built tree [15, 17].

There are various other methods available to find the optimal splitting position, such as using a piecewise quadratic function which uses a small number of cost samples [18]. Such methods will lead to an improvement in tree generation time, but not quality or cost.

The tree generator operates on input from a file created in a commercial modelling suite, and outputs a file ready to be read by the rendering software. Additionally, a visualisation of the tree generated is created.

The *k*D-tree generator was written in Java and executed in version 1.6.0_18 of the Java Runtime Environment. It is acknowledged that Java may not be the most efficient execution platform, however tree generation time is not included in any render times presented during algorithm analysis. Additionally, the execution platform has no impact on the quality or cost of the tree generated.

3.1.2 Bounding Volume Hierarchy Generator

The technique used for BVH construction in the BVH generator is as follows. First, model geometry is recursively divided into octrees (a volume halved in all three axes, resulting in eight distinct sub-volumes). When a sufficiently small number of triangles remain in a voxel (or volume element), these are bounded into an ellipsoid of optimal volume. Only the dimensions of the triangles which are inside the voxel are used to form the ellipsoid, not the dimensions of the voxel itself. This results in ellipsoids of minimum volume. It should be noted that in our chosen generation method, volumes either contain further volumes or triangles, but not both.

The BVH generator, like the *k*D-tree generator, operates on input from a file created in a commercial modelling suite, and outputs a file ready to be read by the rendering software.

The bounding volume hierarchy generator was written in Pascal and compiled using Free Pascal Compiler version 2.0.4.

While we accept that more advanced techniques for BVH construction now exist, including variants making use of SAH based spatial splitting [19, 20], the basic technique used here provides an acceptable alternative to *k*D-trees, which are generally considered the most efficient acceleration structure. We do however recognise that the times presented for the ray processing functions of the renderer could improve with the use of a more recent BVH generation strategy.

3.1.3 Photon Map Renderer

The photon mapping rendering software created as part of this work makes use of methods commonly used for their efficiency to execute the core functions. The software is written in optimised C code and compiled using Borland 5.0 compiler.

Badouels method is used for RTI calculations. There are various alternatives, several of which we have implemented and tested, however we have found little performance difference between them. Our findings are supported by the comparison made by Möller and Trumbore between their own algorithm and Badouels [21].

A balanced *k*D-tree was used for the photon map structure facilitating efficient searching, which is a standard and commonly used approach [2]. Both *k*D-tree and BVH RTI acceleration structures are supported as RAS, with *k*D-trees considered to be the most efficient structure available [15, 22, 23]. Havrans *ArecB* method [15] of *k*D-tree traversal is used.

There are of course many other methods and techniques available for both RTI, RAS and PS processing. A discussion on several of these techniques is presented in section 4.

As already mentioned, our rendering software incorporates several features, as detailed in Table 1, that are over and above those provided by a standard photon map renderer. Some of these functions allow us to accurately analyse the rendering algorithm, while others augment physical realism and therefore improve realism in the images synthesised.

Feature	Description
Algorithm Profiling	To accurately analyse the algorithm, a profiling procedure was built into the rendering software. This records various function call counts and the times spent in various algorithmic components, which are written to file on rendering completion. These statistics allow us to determine which functions were the most used, their average execution times, and the proportion of total execution time for which they are accountable.
Light-Surface Interaction Model using Fresnel laws [24]	The incident angle of light and the refractive index of model surfaces are used to calculate the surface parameters on-the-fly using Fresnel laws. This avoids using fixed proportions for surface absorbance, reflectance and transmittance.
Microfaceting (microscopic surface roughness)	The microfaceting of a surface is used to calculate the direction of rays leaving a surface [25, 26]. These can be either reflected or transmitted rays. A totally microscopically rough surface will be a Lambertian or diffuse reflector; where as a surface with no microfaceting will be a completely specular reflector. This avoids the use of inaccurate fixed diffuse and specular coefficients.

Table 1: Software renderer features.

3.2 Test Model

For simple comparison, all software analysis was performed on a single test scene. Details of the base model are shown in Table 2, with a sample rendering shown in Figure 4. Note that the room scene sits on a plane explaining the larger scales in the *X* and *Z* axes. The room itself is similar in scale in all three axes.

Where models of varying complexity were required, extra furniture and plants were added in and around the scene which had the effect of increasing triangle count while leaving other factors constant.

Property	Value
Triangles	54841
X Scale	5000.000
Y Scale	102.812
Z Scale	5000.000

Table 2: Statistics of base model used during software profiling.

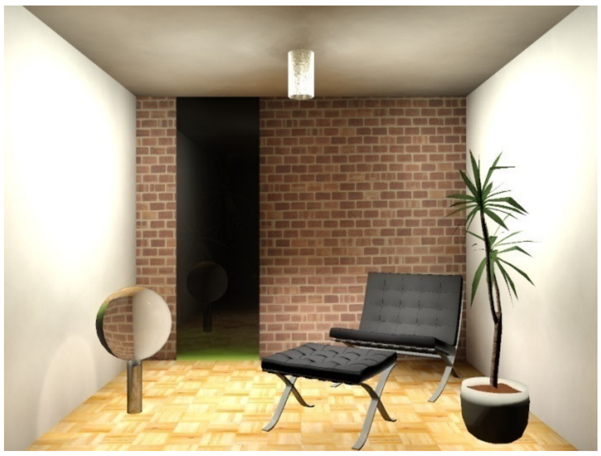


Figure 4: Photon map rendering of test model using standard settings.

3.3 Analysis of RTI Acceleration Structures

In this section, RAS methods of axis-aligned ellipsoid based bounding volume hierarchies and *k*D-trees are analysed. Both structures are generated using the software described in sections 3.1.1 and 3.1.2 respectively and make use of the base model as described in section 3.2.

It should be noted that the time associated with the generation of the RAS structures is not included in rendering timing results. However, we must ensure that renderings make use of the optimal RAS structures which lead to minimal render time. This will allow the accurate quantification of any acceleration achieved. In this section we aim to find the RAS generation settings that allow creation of these optimal structures.

In the case of *k*D-tree generation, the identification of split positions is determined using the upper and lower bounds of each primitive in each axis where there are fewer than *x* primitives in a node, otherwise *x* positions are found from the ordered set of these upper and lower bounds. This option tends towards giving the best quality tree as the number of candidates, *x*, grows. Trees of varying cost are generated using varying node traversal and primitive intersection costs in the SAH calculations. For BVHs, the maximum number of triangles per voxel is varied in order to produce a range of BVHs that vary in the number of triangles per voxel and depth of hierarchy.

The output from this analysis will reveal the settings for both RAS methods that led to the fastest rendering times. The most efficient technique and settings found during this analysis will be used in all further algorithm analysis.

3.3.1 kD-trees

As stated previously, the function used to identify split candidates during analysis is based on the following criteria:

- Subdivision of node volume at the higher and lower bound of each primitive contained in each axis, however if there are greater than *x* primitives in a node, the upper and lower bounds of these primitives are found and ordered. Then, *x* positions are used from this ordered set.

In this analysis, upper and lower boundaries of all primitives are considered as candidates if there are less than 100 primitives in any node.

While primitive intersection and node traversal costs can be measured, it is difficult to attain values for these costs in software with great accuracy due to the variable early exiting possible in both routines. For this reason multiple trees have been generated as these cost arguments were varied, which has resulted in trees of varying depth and quality.

In the following sections we first show the result of varying node intersection cost on overall rendering time. Using the most efficient value indicated for this node traversal cost we then vary triangle intersection cost in the same manner. All other rendering parameters remain constant during these tests. The resultant values for these costs, giving minimal renderer execution time, we consider to be optimal to our test model and rendering setup. These will be used in all further analysis where *k*D-trees are employed.

3.3.1.1 Varying Node Traversal Cost

In this section we present results gained from the rendering of our test model using *k*D-trees for RAS, which have been generated with varying node traversal cost. The actual node traversal costs used in the study range from two orders of magnitude below the triangle intersection cost to two orders of magnitude above.

The rendering results for our test model making use of *k*D-trees generated with varying node traversal cost are shown are shown in Table 3, and graphically in Figure 5.

Node Traversal Cost	Triangle Intersection Cost	Tree Depth	RTI Time (s)	*k*D Traversal Time (s)	RTIs Tested Per Ray	Render Time (s)
100	1	20	114.007	24.136	255.787	238.906
50	1	21	76.628	19.601	174.170	194.547
10	1	27	37.547	11.895	83.914	147.500
5	1	29	26.371	10.610	54.785	134.469
1	1	36	13.372	8.544	27.379	121.750
0.5	1	37	12.653	9.515	24.718	118.110
0.1	1	38	12.466	9.571	23.116	115.750
0.05	1	39	13.300	9.030	23.003	115.234
0.01	1	39	13.473	8.890	22.837	115.235

Table 3: Rendering analysis of test model using *k*D-trees as RAS generated using varying node traversal cost.

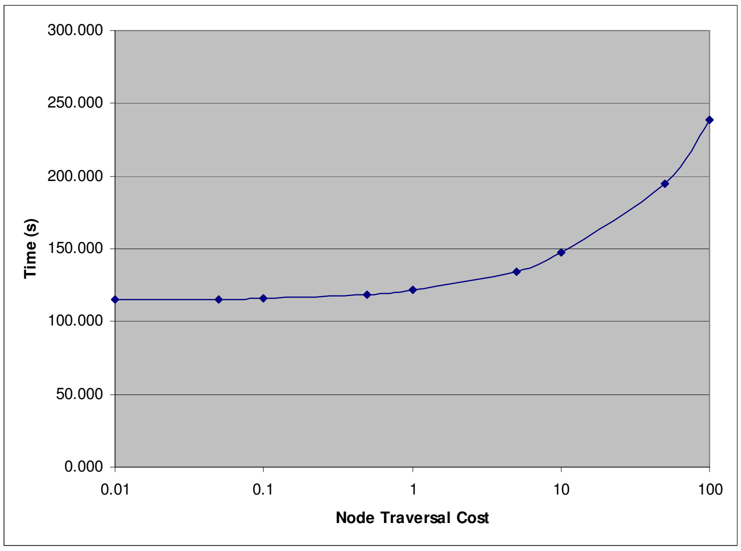


Figure 5: Effect of changing node traversal cost on render time.

We can see that the node traversal costs leading to minimal render time are those having values of 0.5 and lower. In such circumstances, the time consumed by the RTI and RAS functions, which are the only core functions affected by the variation in RAS used, sum to approximately 19% of render time. The remaining render time is consumed in other functions, including the PS core function, however these times remain constant.

We choose to take the optimal value of node traversal cost as 0.5 going into the next test where triangle intersection cost is varied. A comprehensive analysis of these results can be found in section 3.3.1.3.

3.3.1.2 Varying Triangle Intersection Cost

Using the node traversal cost leading to the minimal rendering time as previously found, we then varied the triangle intersection cost in the same manner. Render times from this analysis are shown in Table 4 and Figure 6.

Node Traversal Cost	Triangle Intersection Cost	Tree Depth	RTI Time (s)	*k*D Traversal Time (s)	RTIs Tested Per Ray	Render Time (s)
0.5	100	39	12.293	9.098	22.817	120.109
0.5	50	39	12.363	8.672	22.837	116.266
0.5	10	39	12.255	9.194	22.997	117.078
0.5	5	38	12.617	9.066	23.115	115.938
0.5	1	37	12.653	9.515	24.718	118.110
0.5	0.5	36	14.063	9.031	27.387	117.250
0.5	0.1	29	28.127	9.390	54.893	133.438
0.5	0.05	27	39.794	10.402	83.950	148.547
0.5	0.01	21	79.510	16.951	174.091	193.765

Table 4: Rendering analysis of test model using *k*D-trees generated using varying triangle intersection cost.

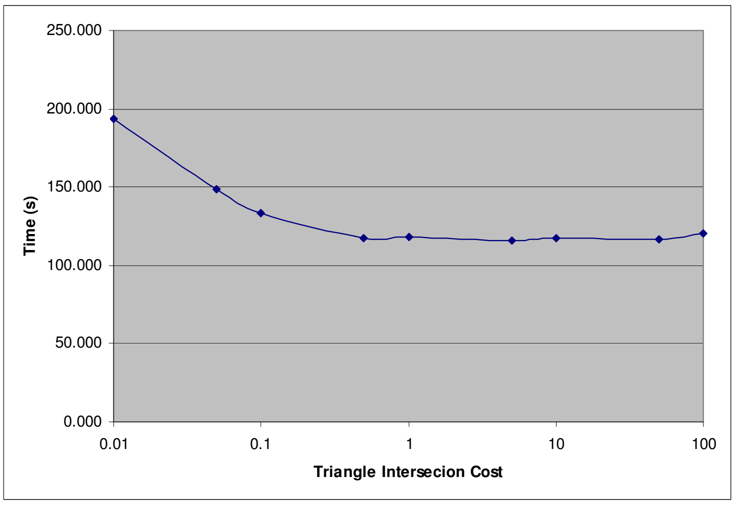


Figure 6: Effect of changing triangle intersection cost on render time.

We can see that the lowest render time occurs when node traversal cost is 0.5 and triangle intersection cost is also set to 0.5 units or greater. Here, the time consumed by the RTI and RAS functions again sum to approximately 19% of render only time, with the remaining render time again consumed in other rendering functions, the execution time of which remains constant.

In the following section these results are analysed before confirming which costs will be taken forward as the most efficient. These costs will be used for *k*D-tree RAS construction during full algorithm profiling.

3.3.1.3 Summary of *k*D-tree Analysis

In the previous two sections several *k*D-trees have been generated and used as RAS in the rendering of our test model.

Findings from this analysis indicate that as node traversal cost increases, the depth of trees generated reduces. This makes sense, as the cost of traversing a ray through many levels of hierarchy with a high associated cost will have a higher overall cost than traversing a ray through fewer levels. As the increase in node traversal cost leads to a reduction in tree depth, the time taken to build the tree also reduces due to the reduced number of split candidates to consider.

It follows that the number of RTI calculations performed per ray increases as more triangles are located within nodes in the reducing number of tree levels – each of which must be tested if a ray intersects one of these nodes, leading to an increase in rendering time.

The opposite is true as triangle intersection cost increases. In this case, it is less costly overall to have more nodes each with fewer triangles than fewer nodes with a greater number of triangles. This is due to the early exiting possible with *k*D-trees, where often the first triangle intersected is the first intersection along the ray. As an increase in triangle intersection cost leads to the generation of trees of increasing depth, build times rise due to the increased number of split candidates that must be considered, however rendering times are reduced.

Figure 7 shows the variation in tree depth as both node traversal and triangle intersection cost are varied.

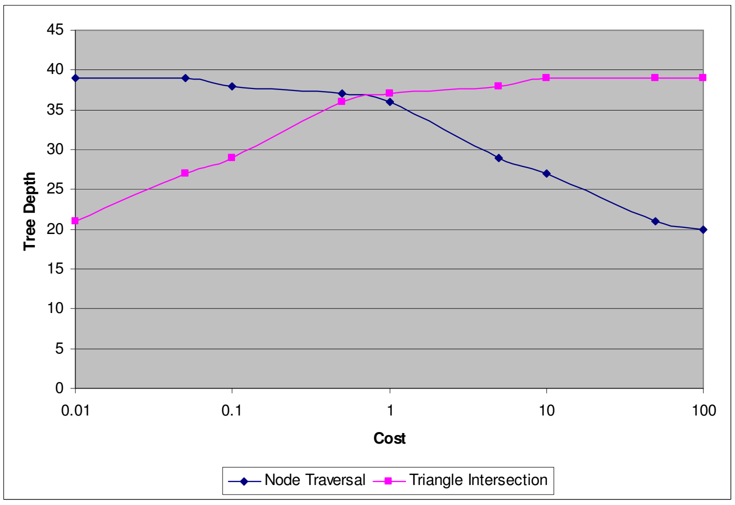


Figure 7: Effect on *k*D-tree depth of node traversal and triangle intersection costs.

To determine the most efficient node traversal and triangle intersection costs going forward, we must look at the total rendering time. Looking at Figure 8, we can see that optimal values for both the node traversal and triangle intersection costs is approximately 0.5. The similarity in these values is not unexpected, as the amount of computation that must be performed for node traversal and triangle intersection operations is comparable.

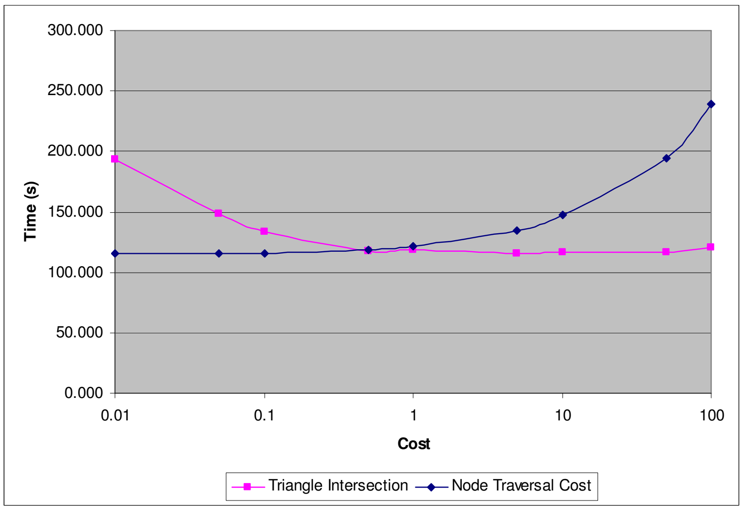


Figure 8: Effect of rendering time of node traversal and triangle intersection costs.

In this section it is also worth discussing the number of RTI calculations performed per ray. Figure 9 shows the number of RTI calculations per ray required in the test renderings, using the trees generated with varying values of node traversal and triangle intersection cost. Note that the optimum lies in the range of 0.5 to 1, confirming our chosen optimal costs.

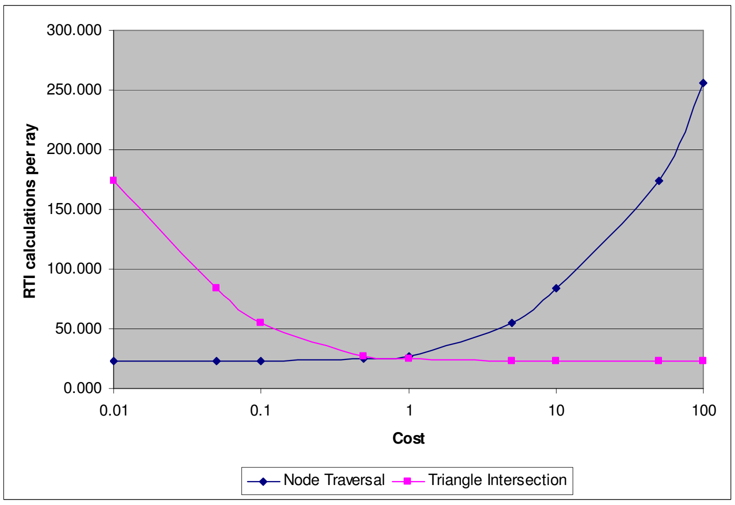


Figure 9: Number of RTI calculations per ray for *k*D-trees generated with varying node traversal and triangle intersection costs.

Havran [15] states that for a well built tree it should be possible to limit the number of RTI calculations per ray to approximately 3. Our tree builder, with its varying split candidate identification methods have been tested using many models and proves that this is correct in some cases.

Interestingly however, for the test model used we believe an optimal value for the number of RTI calculations per ray to be approximately 22. This is the number of RTI calculations processed per ray when overall rendering time is minimal for the various renderings performed using trees generated using varying node traversal and triangle intersection cost.

It should again be noted that our SAH based *k*D-tree generator is based on an outline provided by Havran, and is optimally coded. All RAS structures generated used SAH cost alone as stopping criteria, and were not artificially limited in depth of tree or number of triangles stored per node.

3.3.2 Bounding Volume Hierarchy (Axis-Aligned Ellipsoids)

The other RAS technique analysed was that of the BVH. The volume used was the ellipsoid, which was orthogonally aligned.

The technique used for BVH construction is as follows. First, model geometry is recursively divided into octrees. When a sufficiently small number of triangles remain in a voxel, these are bounded into an ellipsoid of optimal volume. Only the dimensions of the triangles which are inside the voxel are used to form the ellipsoid, not the dimensions of the voxel itself. This results in ellipsoids of minimum volume.

It should be noted that in our chosen generation method, volumes either contain further volumes or triangles, but not both.

3.3.2.1 Maximum Triangles per Volume

Profiling results for renderings of the test model constructed using BVHs with varying maximum triangles per volume are shown in Table 5. Graphical representations of these results are shown in Figure 10.

Maximum Triangles per BV	BVH Depth	RTI Time (s)	BVH Traversal Time (s)	RTIs Tested Per Ray	BVs Tested	Render Time (s)
16	9	638.542	141.333	1773.360	514149541	870.578
32	8	633.161	167.264	1762.114	762488272	888.672
64	8	572.385	210.319	1771.126	515575651	871.218
128	7	649.535	209.450	2022.670	352127240	948.094
256	6	700.960	221.923	2203.074	263171079	1009.375
512	5	814.104	253.851	2575.602	208739948	1157.515

Table 5: Rendering analysis of test model using BVHs generated with varying maximum triangles per volume.

Results show that total execution time rises as the maximum number of triangles allowed per bounding volume (BV) increases. With larger numbers of triangles per volume, all of these triangles must be tested for intersection every time a ray intersects the volume. Volumes with larger numbers of triangles also tend to be larger than those with fewer, making them more likely to be intersected by rays. Both of these facts explain why there is a trend for rendering times to increase with number of triangles per BV.

There will of course be a limit. As the maximum number of triangles per volume falls, more volumes and levels of hierarchy are required. Testing all of these volumes for intersection, each containing few triangles, will be costly. An optimal bounding strategy will therefore be a compromise of the number of volumes and the number of triangles contained per volume. For the test model we consider the optimal maximum triangles per volume to be in the region of 32 to 64.

Like in the *k*D-tree analysis, the render time consumed in functions other than RTI and RAS remains constant.

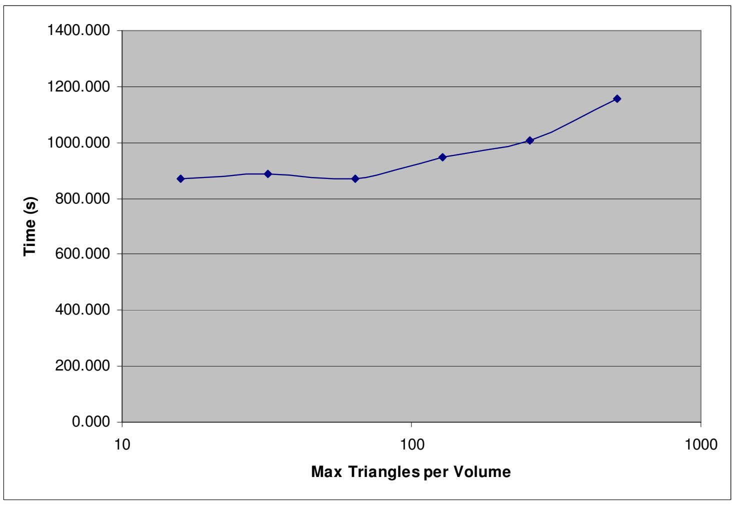


Figure 10: Rendering times versus maximum triangles per volume for test model using BVH as a RAS.

A comparison between the two RAS methods analysed is made in the following section.

3.3.3 Summary of RAS Analysis

The analysis of two common RAS techniques has allowed us to determine the optimal *k*D-tree and BVH generation settings for the test model used in our algorithm analysis.

We have determined that execution times while using BVHs for RAS are significantly longer than when using *k*D-trees. This can be explained by the early exit strategies possible with *k*D-trees, which significantly reduces the number of RTIs which must be computed (see section 4.2.4).

The data presented so far indicates without doubt that BVHs are less efficient than *k*D-trees when both structures are built using the techniques described in section 3.1 for renderings of our test model. As any acceleration hardware will be compared with the most efficient technique, *k*D-trees will be used as the RAS in all further algorithm analysis. These trees will be constructed using the optimal build settings found.

These optimal node traversal and triangle intersection costs will now be held constant for all further tree generation, allowing fair comparison of results in the subsequent further detailed analysis of the photon mapping algorithm itself.

3.4 Analysis of the Photon Mapping Algorithm

Our initial investigations and testing has led us to believe that three algorithmic components can be held accountable for the majority of photon mapping execution time. These are: the processing of RTI calculations, performing any traversal computation needed for the RAS technique employed, and the processing associated with the photon map search function which is used during the calculation of irradiance estimates. In this section we aim to definitively confirm in which functions the majority of algorithm execution time is consumed.

In order to make an informed decision on whether hardware acceleration for any particular function would be a worthwhile investment, we also aim to see how the proportions of time spent in rendering functions varies with changes to rendering parameters. Providing acceleration hardware for a function that is only time consuming in exceptional circumstances would be uneconomical.

The rendering parameters we varied during the algorithm analysis are:

1. Number of photons emitted from scene light sources
2. Photon search radius for photon map search
3. Model complexity

As well as the proportions of execution time spent in the various rendering functions, the actual times expended in these routines are required. This data will feed into the next stage of this work, where we look to accelerate the most time consuming algorithmic components. Using this timing data, we will be able to quantify any acceleration achieved.

3.4.1 *Standard Renderings*

As a baseline for comparison, timing details for renderings of four models, all based on the test scene shown in section 3.2, are presented. All four use *k*D-trees for RAS, using the optimal settings found for tree generation. The rendering settings used are shown in Table 6, while a breakdown of execution time is shown in Table 7 and Figure 12. A visualisation of the RAS employed for one of the models is shown in Figure 11, with red, green and blue lines denoting splits in the *X*, *Y* and *Z* axes respectively, with white lines denoting leaf nodes.

Property	Model 1	Model 2	Model 3	Model 4
Image Size	800 x 600 pixels	800 x 600 pixels	800 x 600 pixels	800 x 600 pixels
Photons Emitted	50,000 photons	50,000 photons	50,000 photons	50,000 photons
Photon Search Radius	20 units	20 units	20 units	20 units
Model Complexity	54,841 triangles	104,381 triangles	151,976 triangles	207,583 triangles
RAS Employed	*k*D-tree with depth of 36 levels, built using node traversal cost of 0.5 units and triangle intersection cost of 0.5 units.	*k*D-tree with depth of 35 levels, built using node traversal cost of 0.5 units and triangle intersection cost of 0.5 units.	*k*D-tree with depth of 35 levels, built using node traversal cost of 0.5 units and triangle intersection cost of 0.5 units.	*k*D-tree with depth of 38 levels, built using node traversal cost of 0.5 units and triangle intersection cost of 0.5 units.

Table 6: Settings used during standard renderings of test models.

	Model 1		**Model 2**		**Model 3**		**Model 4**	
Algorithmic Component	**Time (s)**	**% of Total**	**Time (s)**	**% of Total**	**Time (s)**	**% of Total**	**Time (s)**	**% of Total**
RTI Computation	14.606	10.638	18.197	9.694	22.929	11.542	25.383	12.085
RAS Computation (*k*D-tree traversal)	8.407	6.123	11.547	6.151	12.917	6.502	15.168	7.222
PS Computation	89.961	65.523	131.751	70.185	133.838	67.372	138.914	66.139
Other (rendering main-line)	24.323	17.716	26.224	13.970	28.972	14.584	30.567	14.553
Total	137.297	100.000	187.719	100.000	198.656	100.000	210.032	100.000

Table 7: Breakdown of times consumed in various rendering functions in rendering of four test models.

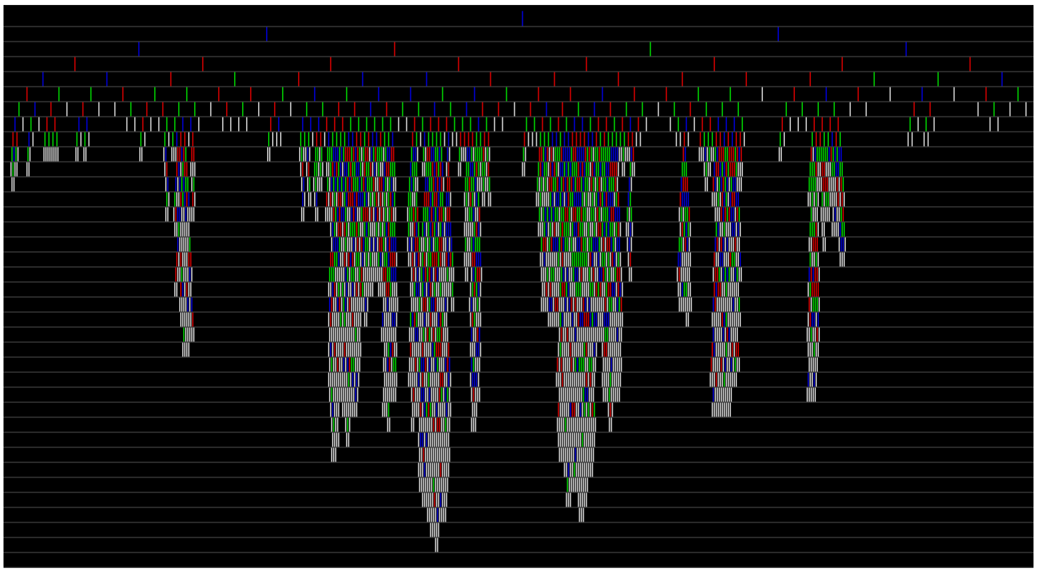


Figure 11: Visualisation of *k*D-tree generated for model 1 containing 54,841 triangles. *k*D-tree built using node traversal and triangle intersection costs of 0.5 units. Red, green and blue lines denote splits in the *X*, *Y* and *Z* axes, with white lines denoting leaf nodes. Lack of resolution means some overlapping in the visualisation at large tree depths.

In the standard rendering setups shown in Table 6, the processing associated with RTI, RAS and PS account for the majority of total rendering time. We therefore refer to the RTI, RAS traversal and PS functions as the core rendering functions in the remainder of this work. The time not consumed by these functions, but spent performing other ancillary rendering tasks is referred to as the main-line rendering time.

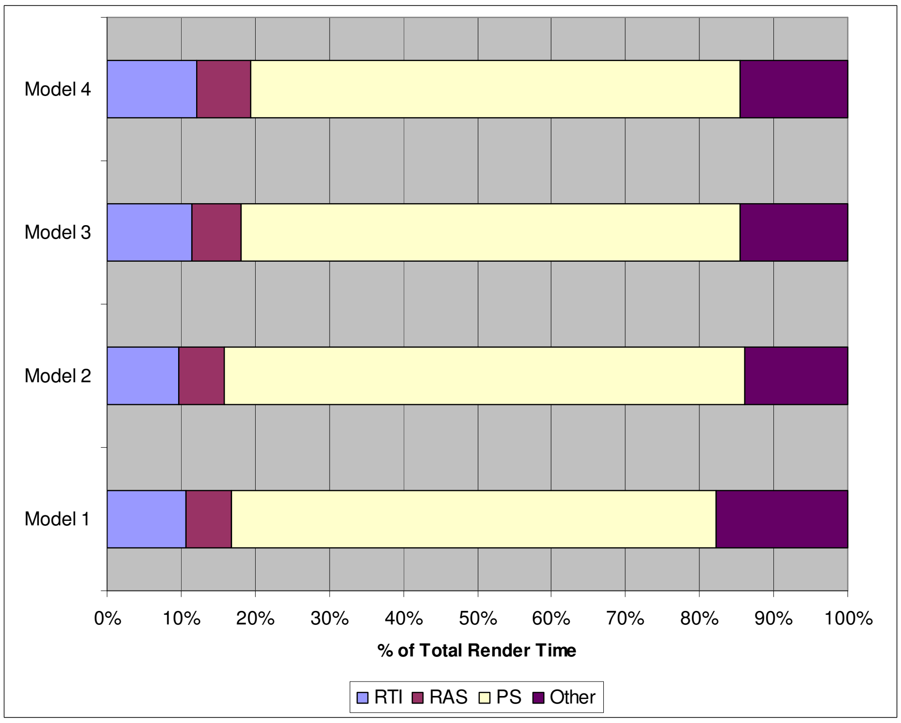


Figure 12: Graphical representation of times consumed by various rendering functions for four standard renderings.

Figure 12 shows that a large proportion (>10%) of total overall rendering time is consumed in main-line rendering tasks for the four standard renderings. In section 3.1.3 we discussed the extra features of our renderer, which are over and above what a standard photon map renderer offers. In order to maintain a fair test, any time spent in these additional functions should be factored out if possible. A breakdown of the time spent in non-core rendering tasks in the four renderings is shown in Figure 13 .

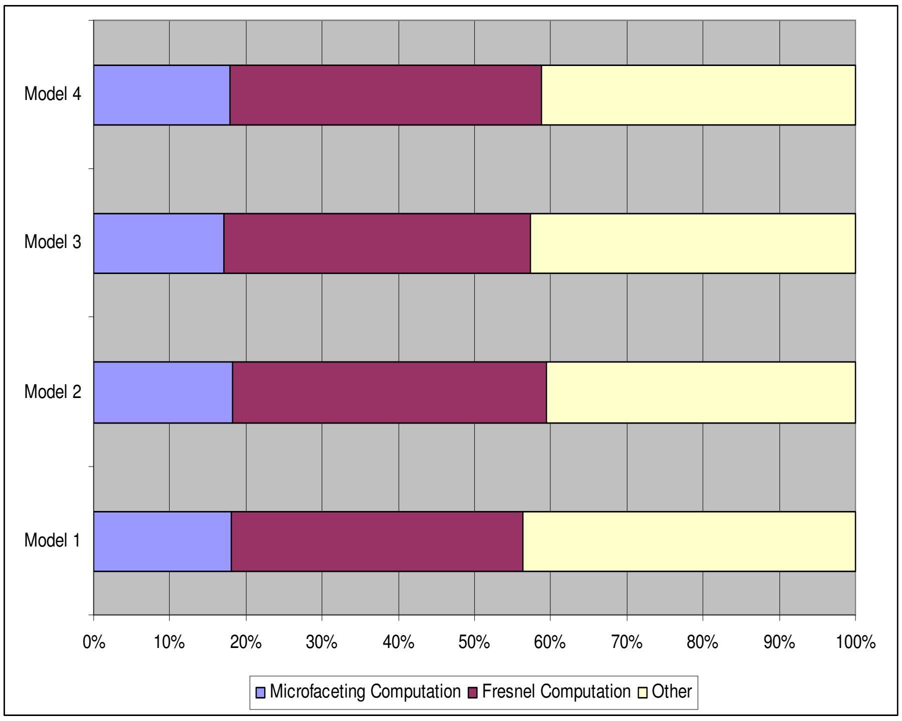


Figure 13: Breakdown of non-core rendering time.

We can see that approximately 50-60% of non-core rendering time in each rendering is consumed in microfaceting and Fresnel (reflection model) calculations, with the remainder being made up in functions handling texture mapping, *k*D-tree balancing and PS density calculations among others.

Both microfaceting and Fresnel functions have been added to improve realism, however neither is provided by a basic photon map renderer. As this is the case, the time spent in these functions should not be considered in any comparisons and be factored out as far as possible. This is a simple task, as these functions can bypassed, with their inputs being reused as reflective, absorptive and transmissive coefficients during ray handling. No alternative processing is required.

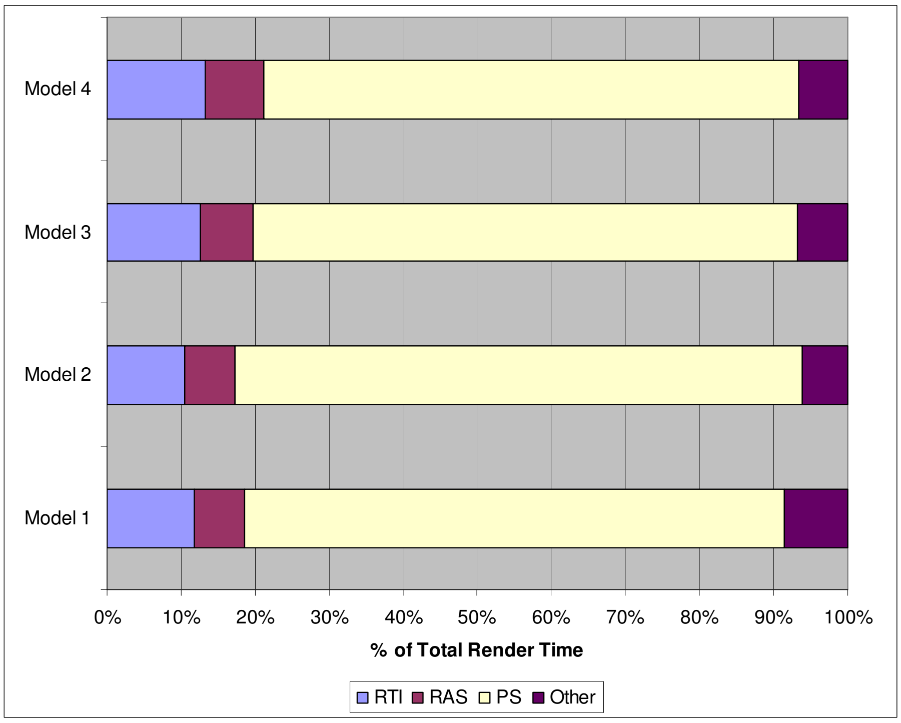


Figure 14: Representation of times consumed by various rendering functions, excluding the time spent in microfaceting and Fresnel calculations in the main-line processing.

Updated results, taking into account the removal of these extra functions are shown in Figure 14. Note the reduction in time spent in main-line rendering functions.

These additional functions, not provided in standard photon map renderers, will be factored out throughout the remainder of our analysis of the photon mapping algorithm.

3.4.2 Varying Common Rendering Parameters

Multiple renderings were performed to ascertain how both time and proportion of time spent in the core rendering functions vary with changes to main rendering parameters. In each of these renderings one or more variables were changed within reasonable limits.

As already mentioned, the parameters varied were:

1. Number of photons emitted from scene light sources
2. Photon search radius for photon map search
3. Model complexity

An analysis of renderings for various image sizes is not presented here; however testing has shown that the percentage of execution time spent in all core functions remains constant with increase in rays fired.

This makes sense, as for every ray the same procedure must be followed, i.e.:

- Traverse RTI acceleration structure
- Test ray for intersection for intersection with triangles
- If appropriate, calculate irradiance estimate making use of photon search function

Therefore, even though more rays are processed, which results in a longer execution time by each core function, the proportion of total execution time used by these core functions remains constant.

The other rendering parameters, photons emitted, search radius and model complexity, were varied in a combined procedure. The former two were varied in order to sustain a sufficient number of photons per search in order to produce a statistically representative estimate of illumination, as is the case in practical rendering. The variations in photons emitted and search radius were repeated in the rendering of four models of varying complexity, as used in the standard renderings. All models were based on the same scene, with triangle count for the more complex models increased by placing extra plants and furniture in and around the room. During the renderings all other rendering parameters remained constant, and are those defined for the standard renderings.

Timing results for these renderings are shown in Table 8, and graphically in Figure 15.

Model Complexity (triangles)	Photon Search Radius	Photons Emitted	RTI Time (s)	*k*D-Tree Traversal Time (s)	Photon Map Search Time (s)	Total Photons Stored	Photons Found per Search	Render Time (s)
55k	10	100k	16.040	10.060	76.023	131580	75.408	114.445
55k	20	50k	14.606	8.407	89.961	65756	85.686	123.576
55k	30	10k	13.486	8.405	65.802	12978	77.511	96.794
104k	10	100k	20.624	13.128	121.035	141918	86.980	172.120
104k	20	50k	18.197	11.547	131.751	71049	90.444	139.321
104k	30	10k	18.694	10.597	99.339	13859	87.309	177.604
152k	10	100k	25.418	13.985	123.447	141951	84.085	149.396
152k	20	50k	22.929	12.917	133.838	70497	88.909	180.949
152k	30	10k	23.012	12.436	102.610	14134	85.845	192.070
208k	10	100k	26.411	16.756	124.468	142532	81.220	114.445
208k	20	50k	25.383	15.168	138.914	70583	86.710	123.576
208k	30	10k	23.848	15.807	104.060	13900	82.703	96.794

Table 8: Details of renderings for varying search radius, number of photons emitted and model complexity. Time consumed in additional rendering functions such as microfaceting and Fresnel calculations has been removed.

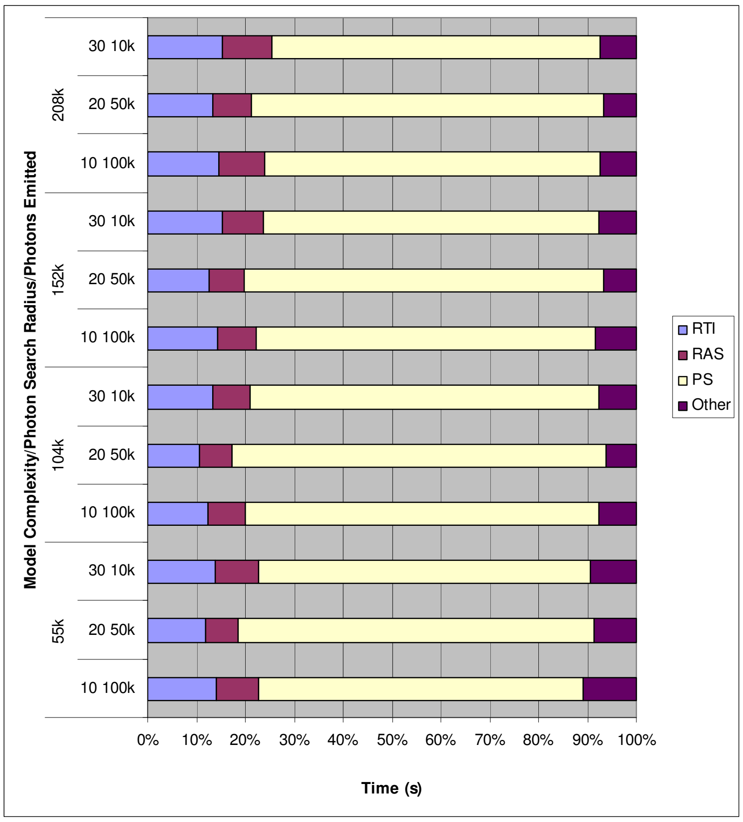


Figure 15: Breakdown of rendering times by function for changes to search radius, photons emitted and model complexity. Time consumed in additional rendering functions such as microfaceting and Fresnel calculations has been removed.

We will first look at the effect of varying the number of photons emitted into the scene. With an increase in photons emitted we see an increase in the number of photons stored, which is expected as more photons will intersect reflective model surfaces. Without any variation in search radius, this increase would lead to a rise in photon search times, as an increased number of photons must be considered during each search.

As photons are essentially rays, with any increase there will be an impact on the ray processing times. In our results we a small increase in both RAS and RTI execution times, which contributed to the rise in overall execution time as the number of photons emitted increased.

Turning to the variation in the photon search radius, we would expect that as the photon search radius increases there would be an increase in photon search execution time. This is typical, as with a greater search volume it is more likely that more photons must be considered for inclusion during each search. This was however counteracted in the results presented by the varying number of photons emitted, which was increased for small search radii, and reduced for larger radii in order to maintain a reasonable number of photons per search to create a statistically representative illumination estimate. It should be noted that *k*NN searching was implemented in our software renderer, however the value of *k* was set to 100 meaning that its use was extremely rare, with the average number of photons found per search being 84.401. As variations in the search radius have no effect on the number of rays fired, there is no effect on RTI and RAS execution times.

The final rendering parameter we look at is model complexity, or the number triangles used in a model description. All rendered models were based on the same design, with triangle count for the more complex models increased by placing extra plants and furniture in and around the room as already stated. The RAS method used, and the settings used for construction of the *k*D-tree itself were kept constant.

We see that an increase in model complexity results in an increase in RTI and RAS time, which is as expected as more triangles must be tested for intersection. There is also a smaller increase in PS time. This stems from the fact that stored photons will be distributed amongst the increased number of reflective surfaces for models containing more objects. Even though a consistent number of photons are stored, this distribution means that there is a reduced chance of no or few photons being found within the search volume. Therefore photons are tested in a greater proportion of searches which leads to a small increase in PS time. If instead of augmenting model triangle count with extra objects, the existing scene was tessellated to introduce a greater number of triangles, we would see no change in PS time.

The results presented here confirm that the vast majority of overall rendering time is spent in the RTI, RAS and PS functions, with only a small proportion spent in other ancillary main-line rendering tasks.

3.4.3 Summary of Algorithm Analysis

The aim of performing this detailed analysis of the photon mapping algorithm was to definitively confirm the main functions that contributed to overall execution time and find typical processing times for these functions. To be able to decide if acceleration of any of these functions was worthwhile, we also sought statistics on how the individual execution times of these functions and their proportion of total render time change as common rendering parameters are varied within reasonable limits.

We were able to confirm that three functions consumed the vast majority of total execution time for all renderings. These are RTI and RAS, which are used to find the first triangle intersected by any particular ray, and PS, which is used to find photons within a particular volume and is used in for illumination calculations. For renderings of our test model, these three functions accounted for over 89.2% of total render time in all cases, and on average 92.1% across all rendering analysis presented. Reducing the execution time of each to negligible levels would result in over an order-of-magnitude speedup in overall render time.

For the model and settings used we found that RTI accounts for approximately 13.5% of total execution time, with RAS processing taking about 8.1% and PS on average 70.6%. This makes both the functions dealing with ray-triangle intersection, RTI and RAS, and also the photon searching routine ideal candidates for acceleration.

In the following section we look at common software methods used in the three core functions to establish if any are suitable for direct hardware implementation, or whether a more novel approach will be required to achieve the acceleration sought.

After designing hardware to speed up these functions, we will need to quantify the acceleration achieved. Some useful statistics gained from the standard renderings can be used in this evaluation. These are presented in Table 9 below.

Property	Model 1	Model 2	Model 3	Model 4
Total RTI + RAS Time (s)	23.013	29.744	35.846	40.551
Time to process a ray (μs)	7.905	9.272	10.386	11.036
Single RTI Time (μs)	0.288	0.305	0.277	0.308
Total PS Time (s)	89.961	131.751	133.838	138.914
Single PS Time (μs)	39.211	52.124	49.516	48.527

Table 9: Statistics from standard renderings of test models.

4 A COMPREHENSIVE ANALYSIS OF COMMON RENDERING FUNCTIONS

In the previous section we confirmed the identity of three rendering functions where the majority of photon mapping execution time is spent. These are:

1. Ray-triangle intersection processing (RTI)
2. Processing of RTI acceleration structures (RAS)
3. Photon map searching (PS)

In this section, we will look at several common methods for performing each of these core tasks.

4.1 Ray-Triangle Intersection (RTI)

There are many ways to test for intersections between triangles and rays. Some of these rely on finding the intersection point on the plane of the triangle first, before testing whether this is inside the triangle [27], while others calculate the intersection directly. In this section we give an overview of a selection of common RTI methods.

The RTI methods investigated are:

- Angle Summation Method
- Edge Crossings Method
- Half-Plane Testing Method
- Snyder-Barr Method
- Badouels Method
- Möller-Trumbore Method
- Plücker Coordinate Method
- Arenbergs Method

Several of these methods are commonly used in rendering solutions due to their efficiency when executed on floating-point platforms. With various number systems and representations available for use in any custom hardware implementation, we also look at methods that are less commonly used, or often disregarded, due to their less efficient execution on floating-point platforms.

4.1.1 *Secondary Methods*

Several of the RTI methods investigated make use of the ray-plane intersection test in order to find the ray-plane intersection distance, and a barycentric coordinate test, which is used to determine whether a point on a plane is within a triangle. An overview of these secondary methods is provided in the following sections.

4.1.1.1 Ray-Plane Intersection Method

The ray-plane intersection method can be used to calculate the distance between a ray origin and the plane on which a triangle lies.

The theory of this method is as follows. A plane can be defined by its normal, *N* and any point on its surface *P*. If any other point *Q* lies on the surface, the equation *N·(P-Q) = 0* will be satisfied as the dot product of perpendicular vectors is always zero. The coordinates of an intersection between a ray with origin *O* and direction *D* and a plane can be represented as *r(t)=O+tD*, where *t* is the distance travelled along the ray in direction *D* to the point of intersection. The ray-plane intersection distance can therefore be calculated as follows:

$$\begin{aligned} &N \cdot P - r(t) = 0 \\ &N \cdot P - (O + tD) = 0 \\ &t = \frac{N \cdot (P - O)}{N \cdot D} \end{aligned} \tag{1}$$

Given $N=[N_x, N_y, N_z]$, $P=[P_x, P_y, P_z]$, $O=[O_x, O_y, O_z]$ and $D=[D_x, D_y, D_z]$, the equation can be expanded, giving:

$$t = \frac{N_X.(P_X - O_X) + N_Y.(P_Y - O_Y) + N_Z.(P_Z - O_Z)}{N_X.D_X + N_Y.D_Y + N_Z.D_Z} \tag{2}$$

To reduce the amount of data needed to describe the plane, *N* and *P* can be combined to form the plane equation [*A, B, C, D*], as shown in equation 3. This can be pre-calculated, reducing storage and computation during rendering.

$$\begin{aligned} &A = N_X \\ &B = N_Y \\ &C = N_Z \\ &D = -(N_X.P_X + N_Y.P_Y + N_Z.P_Z) \end{aligned} \tag{3}$$

The resulting ray-plane intersection distance calculation becomes:

$$t = \frac{-(A.O_X + B.O_Y + C.O_Z + D)}{A.D_X + B.D_Y + C.D_Z} \tag{4}$$

The actual coordinates of the intersection can be found by substituting this distance into the ray equation.

4.1.1.2 Barycentric Coordinate Test

Some RTI methods use the barycentric coordinate point-in-triangle test to determine whether a point on the plane of a triangle is within the bounds of the triangle itself.

The theory of the point-in-triangle test using barycentric coordinates is as follows. Any point *P* on the plane of triangle *ABC* can be described by:

$$P = A + \alpha\overrightarrow{AB} + \beta\overrightarrow{AC} \tag{5}$$

$$P = A + \alpha(B - A) + \beta(C - A) \tag{6}$$

$$P = (1 - \alpha - \beta)A + \alpha B + \beta C \tag{7}$$

The barycentric coordinates of the point *P* are $(1 - \alpha - \beta) = \lambda$, α and β. These coordinates are proportional to the areas of triangles *PBC*, *PCA* and *PAB*.

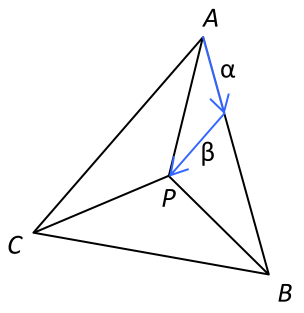


Figure 16: Barycentric coordinates illustration.

For the point *P* to lie within triangle *ABC*, $0 \leq \lambda, \alpha, \beta \leq 1$ and $\lambda + \alpha + \beta = 1$ must hold.

4.1.2 Angle Summation Method

A simple angle summation test can be used to test whether a point on a plane is within a triangle [28]. Vectors from the intersection point to all vertices are formed, and if the angles between these vectors sum to 360° then the point is within the triangle. The intersection point is required, which can be calculated using the ray-plane intersection method.

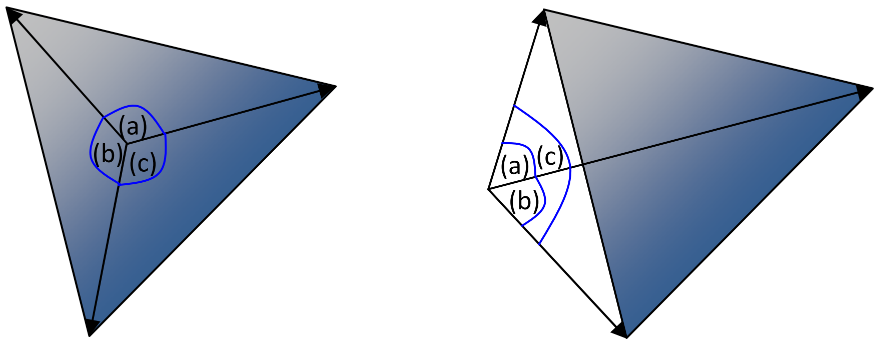


Figure 17: Angle summation illustration. Point is within triangle if angles between vectors sum to 360° (left). Point is outside triangle if angles between vectors sum to less than 360° (right).

A common issue with this method is in the comparison of the sum of the angles, which can be found using a simple dot product, with 360°. Rounding errors in the calculations mean that in practice a small epsilon value must be used in the comparison.

4.1.3 Half Plane Testing Method

Half plane testing is another point in triangle test and relies on the pre-computation of the ray-plane intersection coordinates.

The theory of the half plane testing method is as follows. The cross product of any two vectors will return a vector that is perpendicular to both. The cross product follows the right-hand rule, meaning that this perpendicular vector changes direction depending on the orientation of the two input vectors.

If a point lies on the same side of each edge of a triangle, then the point is within the triangle. This can be tested using the cross products of the vectors between each vertex and the intersection point, and the triangles edges. If all resultant vectors have same direction, the point is within the triangle [29, 30].

As both the vertex-intersection point vectors and the triangle edges lie on the plane of the triangle, the vector results of the cross products will be in either in the same direction as the plane normal, or in the opposite direction. The directions can be tested using either sign checking or dot products with the plane normal.

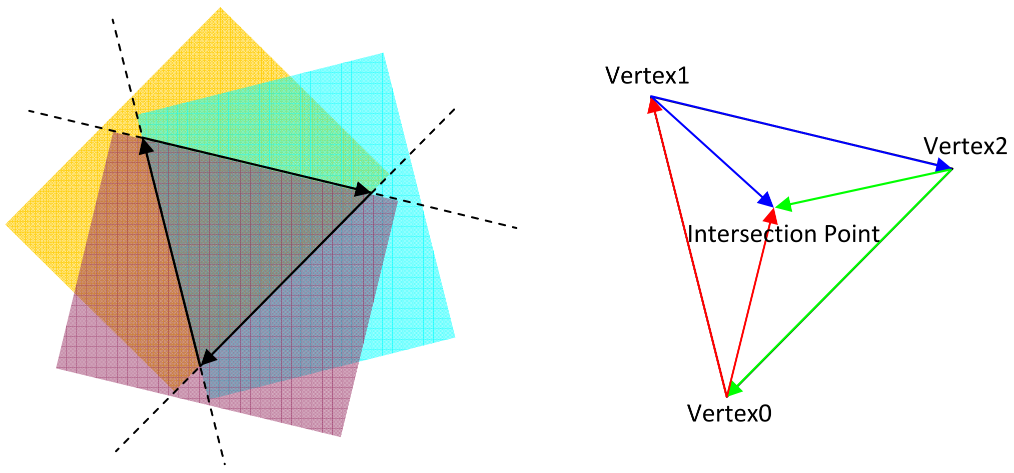


Figure 18: Half Plane Testing illustration. Area of point inclusion for each half plane test (left). Cross products of coloured vector pairs will have same direction if point is within triangle (right).

4.1.4 Edge Crossings Method

To calculate whether a ray intersects a triangle using the edge crossings method [31, 32], two components of the intersection coordinates are required. These components can be found using the ray equation and the previously calculated ray-plane intersection distance.

To perform the RTI test, the triangle is first projected onto a primary plane (*XY*, *XZ* or *YZ*) which provides the largest projection area. To test whether a point is within the triangle, a ray is cast from the intersection point, which has also been projected onto the same plane, along one of the plane axes. The number of triangle edge crossings are counted. If there are no crossings, then the ray has missed all edges and the point is outside the triangle. If there are two edge crossings, then the ray has crossed two edges and again the test point is outside the triangle. The point is only inside the triangle if there is one and only one edge crossing.

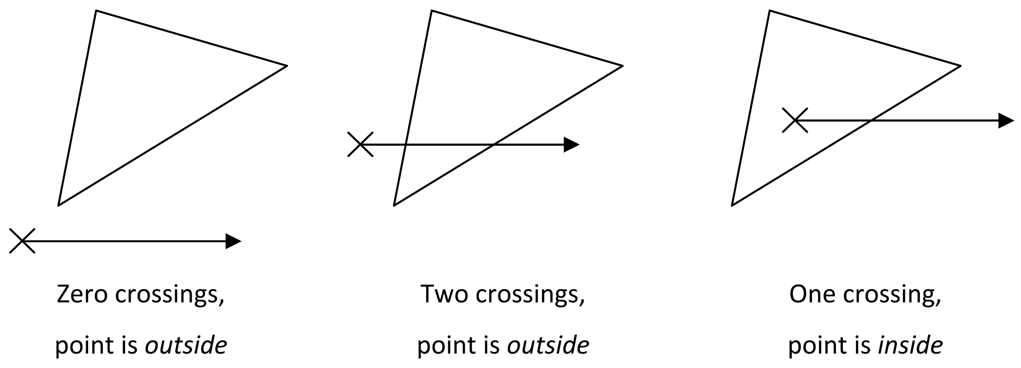


Figure 19: Edge Crossings illustration. Zero line crossings, point is outside triangle (left). Two line crossings, point is outside triangle (centre). One line crossing, point is inside triangle (right).

4.1.5 Snyder-Barr Method

In the Snyder-Barr method [33], triangles are again projected onto a primary plane, *XY, XZ or YZ,* that is perpendicular to the dominant axis of the triangle normal, *N*. This ensures that the area of projection is maximised. The dominant axis, *i₀*, is determined as follows:

$$i_0 = \begin{cases} 0 & if \left|N_X\right| is \max \\ 1 & if \left|N_Y\right| is \max \\ 2 & if \left|N_Z\right| is \max \end{cases} \tag{8}$$

To perform the RTI test, the intersection coordinates are calculated in the two projection axes, i_1 and i_2, (i_1, i_2 ∈ discuss), which are different from i_0 and are unequal. This calculation takes the ray-plane intersection distance as an input, which must be pre-calculated using the ray-plane intersection method.

For any point *P* on the plane of a triangle *ABC*, the three barycentric coordinates of *P* can be calculated as shown in equations 9-11. The edge data required can be pre-calculated and stored in each case, reducing runtime workload.

$$\lambda = \frac{[(C-B)\times(P-B)]_{i_0}}{N_{i_0}} \tag{9}$$

$$\alpha = \frac{[(A-C)\times(P-C)]_{i_0}}{N_{i_0}} \tag{10}$$

$$\beta = \frac{[(B-A)\times(P-A)]_{i_0}}{N_{i_0}} \tag{11}$$

To test whether the intersection point is within the triangle, these values are tested using the barycentric coordinate test.

4.1.6 Badouels Method

Badouels RTI method [34] is another which uses the barycentric coordinate test to determine whether a point lies within a triangle. Like Snyder-Barr, this method relies of the pre-calculation of the distance from ray origin to the plane of the triangle using the ray-plane intersection method.

The theory behind the method is as follows. Any point *P* on the plane of the triangle *ABC* can be expressed as:

$$P = A + \alpha(B - A) + \beta(C - A) \tag{12}$$

The three components of this equation are:

$$\begin{aligned} P_X - A_X &= \alpha(B_X - A_X) + \beta(C_X - A_X) \\ P_Y - A_Y &= \alpha(B_Y - A_Y) + \beta(C_Y - A_Y) \\ P_Z - A_Z &= \alpha(B_Z - A_Z) + \beta(C_Z - A_Z) \end{aligned} \tag{13}$$

Much like the Snyder-Barr method, the triangle is projected onto one of the primary planes, *XY*, *XZ* or *YZ*, which is perpendicular to the dominant axis of the triangle normal. The dominant axis i_0 is determined in the same way, with the indices i_1 and i_2 representing the projection plane being different from i_0 and unequal.

The vectors *u* and *v* are formed using the projected vectors *P-A*, *B-A* and *C-A*, as shown in equation 14 below.

$$\begin{aligned} u_0 &= P_{i_1} - A_{i_1} \quad u_1 = B_{i_1} - A_{i_1} \quad u_2 = C_{i_1} - A_{i_1} \\ v_0 &= P_{i_2} - A_{i_2} \quad v_1 = B_{i_2} - A_{i_2} \quad v_2 = C_{i_2} - A_{i_2} \end{aligned} \tag{14}$$

As $u_{1,2}$ and $v_{1,2}$ are not ray dependant, they can be pre-calculated and stored along with the index values $i_{1,2}$, leaving only u_0 and v_0 to be calculated at runtime. This allows equation 13 to be reduced to:

$$\begin{aligned} u_0 &= \alpha u_1 + \beta u_2 \\ v_0 &= \alpha v_1 + \beta v_2 \end{aligned} \tag{15}$$

The values of α and β, required for the barycentric test, are found as follows:

$$\alpha = \frac{\begin{vmatrix} u_0 & u_2 \\ v_0 & v_2 \end{vmatrix}}{\begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix}} \quad \beta = \frac{\begin{vmatrix} u_1 & u_0 \\ v_1 & v_0 \end{vmatrix}}{\begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix}} \tag{16}$$

As the divisor is not ray dependent, this value (or its reciprocal) can be pre-calculated and stored, further reducing runtime workload.

4.1.7 Möller-Trumbore Method

The algorithm presented by Möller and Trumbore [35] does not make use of the intersection coordinates on the plane of a triangle. Instead, the origin of the ray is transformed giving a vector containing the ray-plane intersection distance and the barycentric coordinates of the intersection.

The theory behind the algorithm is as follows. The point of intersection of a ray *r(t)=O+tD*, and the triangle *ABC* gives $O+tD=\lambda A+\alpha B+\beta C$, which can be rearranged to form:

$$[-D,(B-A),(C-A)]\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix}=O-A \tag{17}$$

This can be solved using Cramer's rule. With $E_1 = B - A$, $E_2 = C - A$ and $T = O - A$, the equation becomes:

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix}=\frac{1}{[-D,E_1,E_2]}\begin{bmatrix} |T,E_1,E_2| \\ |-D,T,E_2| \\ |-D,E_1,T| \end{bmatrix} \tag{18}$$

With the reusable factors $F = (D \times E_2)$ and $G = (T \times E_1)$, this can be rewritten as:

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix}=\frac{1}{(D\times E_2)\cdot E_1}\begin{bmatrix} (T\times E_1)\cdot E_2 \\ (D\times E_2)\cdot E_1 \\ (T\times E_1)\cdot D \end{bmatrix}=\frac{1}{F\cdot E_1}\begin{bmatrix} G\cdot E_2 \\ F\cdot T \\ G\cdot D \end{bmatrix} \tag{19}$$

To determine whether the ray intersects the triangle, the barycentric test is used on the returned values. The distance from ray origin to triangle is also returned, however the intersection coordinates are not. These must be subsequently calculated by substituting this distance into the ray equation if needed.

4.1.8 Plücker Coordinate Method

Plücker coordinates [36-39] can be used to specify lines and rays in 3D space using six dimensional (6D) vectors [*L1, L2, L3, L4, L5, L6*].

The Plücker coordinates of an oriented line passing through two points are [*ax-bw, ay-cw, by-cx, az-dw, bz-dx, cz-dy*], where the two points have Cartesian coordinates of [*b/a, c/a, d/a*] and [*x/w, y/w, z/w*], and the line passes through them in this order. The homogenous coordinates of these points are [*a, b, c, d*] and [*w, x, y, z*].

The Plücker coordinates of a line always satisfy the following equation:

$$L1*L6-L2*L5+L3*L4=0 \tag{20}$$

Two oriented lines can interact in three different ways. They can intersect, or one can go either clockwise or anti-clockwise around the other. These three cases are depicted in Figure 20 for oriented lines *L* and *M*. The perspective given is that of looking along line *M*.

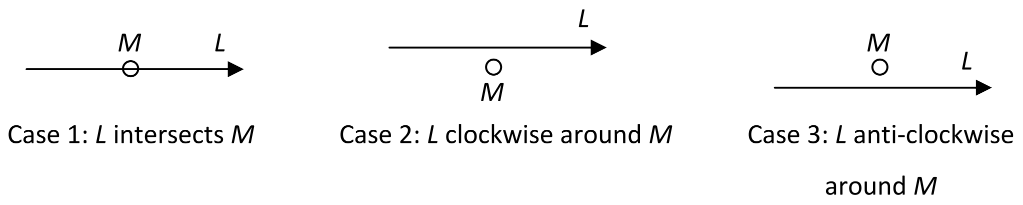

M
L
Case 1: L intersects M
L
M
Case 2: L clockwise around M
M
L
Case 3: L anti-clockwise around M

Figure 20: Plücker coordinate illustration. Three cases of oriented line interactions. The perspective given is that of looking along line *M*.

Using the Plücker coordinates of the two lines *L* and *M*, it is possible to discover which of these cases applies. For this, the permuted inner product of the lines is used. Equation 21 shows the case when *L* intersects *M*, equation 22 shows the case when *L* is clockwise around *M* and equation 23 is the case when *L* is anti-clockwise around *M*.

$$L1M6-L2M5+L3M4+L4M3-L5M2+L6M1=0 \tag{21}$$

$$L1M6-L2M5+L3M4+L4M3-L5M2+L6M1<0 \tag{22}$$

$$L1M6-L2M5+L3M4+L4M3-L5M2+L6M1>0 \tag{23}$$

To test whether a ray intersects a triangle, the Plücker coordinates of the ray and the three triangle edges can be found and tested using the above formula. The ray intersects the triangle only if it hits one of the triangle edges, or it is either clockwise or anti-clockwise around all three edges. This is shown in Figure 21.

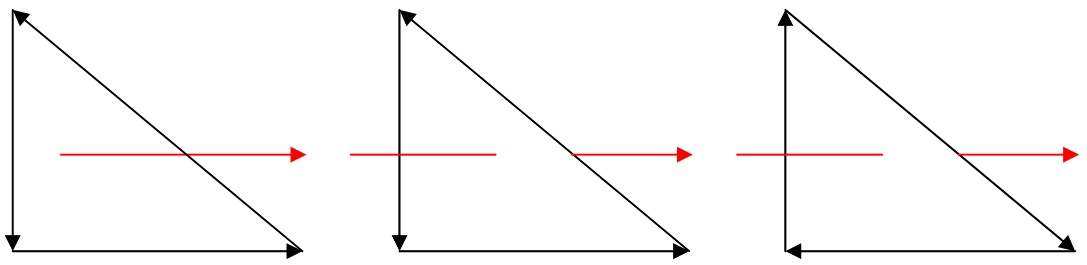


Figure 21: Plücker Coordinate example. Ray intersects one edge (left), ray is clockwise around all triangle edges (centre), ray is anti-clockwise around all triangle edges (right).

The Plücker coordinates of the three triangle edges can be pre-computed to reduce render time workload. For the triangle *ABC*, the Plücker coordinates for its three edges, *E0*, *E1* and *E2*, are shown in equation 24 below.

$$\begin{aligned}
E0[0] &= A_xB_y - B_xA_y\\
E0[1] &= A_xB_z - B_xA_z\\
E0[2] &= A_x - A_y\\
E0[3] &= A_yB_z - B_yA_z\\
E0[4] &= A_z - B_z\\
E0[5] &= B_y - A_y\\[6pt]
E1[0] &= B_xC_y - C_xB_y\\
E1[1] &= B_xC_z - C_xB_z\\
E1[2] &= B_x - B_y\\
E1[3] &= B_yC_z - C_yB_z\\
E1[4] &= B_z - C_z\\
E1[5] &= C_y - B_y\\[6pt]
E2[0] &= C_xA_y - A_xC_y\\
E2[1] &= C_xA_z - A_xC_z\\
E2[2] &= C_x - C_y\\
E2[3] &= C_yA_z - A_yC_z\\
E2[4] &= C_z - A_z\\
E2[5] &= A_y - C_y
\end{aligned} \tag{24}$$

Data stored per triangle can be reduced by noting that *E2*[2], *E2*[4] and *E2*[5] are not needed, as shown below in equation 25.

$$\begin{aligned}
E0[2] + E1[2] + E2[2] &= A_x - B_x + B_x - C_x + C_x - A_x = 0\\
E0[4] + E1[4] + E2[4] &= A_z - B_z + B_z - C_z + C_z - A_z = 0\\
E0[5] + E1[5] + E2[5] &= B_y - A_y + C_y - B_y + A_y - C_y = 0\\[6pt]
E2[2] &= -E0[2] - E1[2]\\
E2[4] &= -E0[4] - E1[4]\\
E2[5] &= -E0[5] - E1[5]
\end{aligned} \tag{25}$$

4.1.9 Arenbergs Method

In Arenbergs method [40], a triangle is transformed so that two of its edges lie on the *X* and *Y* coordinate axes and are scaled to unit length, with the normal of the triangle lying on the *Z* axis. This results in a right-angled triangle with vertices at (0,0,0), (1,0,0) and (0,1,0), as shown in Figure 22.

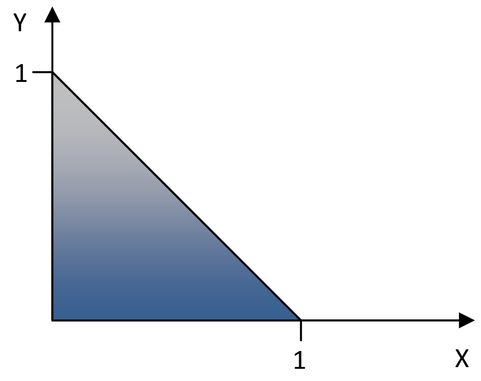


Figure 22: Arenbergs method illustration. Triangle transformed to unit space.

To perform the intersection test, both ray origin and direction are subjected to the same transformation. The division of the transformed *Z* direction and origin components of the ray returns the ray-plane intersection distance (which is equal in original and transformed space).

The *X* and *Y* components of the intersection point, in original space, can be found using the original ray equation and the returned value of *t*. These values are then themselves transformed, so that a simple comparison can be made to determine if the intersection point is within the triangle in transformed space ($x > 0; y > 0; x + y \leq 1$).

4.1.10 A Discussion on RTI Methods

We have presented several different RTI methods, all varying in the amount of processing involved and the input data required.

From these methods, Arenbergs method (or modified versions) and the Möller-Trumbore method are the most commonly used, due to their efficient processing on floating-point platforms. It is these two methods that the majority of the recent ray processing architectures use.

There are drawbacks to the use of certain algorithms, such as the rounding-errors associated with the angle summation technique. Problems have also been identified in Arenbergs method, where a loss of precision due to the transformation of ray and triangle to unit triangle space can cause visible errors in images generated [41].

In the course of our testing, many different RTI methods have been implemented. We have found little variation in performance between most well known and commonly used techniques.

4.2 RTI Acceleration Structures (RAS)

We have already determined that the execution times of rendering methods such as ray tracing and photon mapping are critically dependant on RTI processing time. The use of the most efficient RTI testing method alone is not enough in most situations and further action must be taken to reduce the number of RTI calculations to perform. Glassner [42], Jansen [43], Havran [15] and Szirmay-Kalos et al [23] provide some useful background on various spatial subdivision methods and types of acceleration data structure.

In this section we give an overview of several common RAS techniques.

4.2.1 Bounding Volumes

Bounding volumes allow model primitives to be bounded within larger objects, and are an effective way to reduce the number of RTI tests to perform. It is a fact that if a ray does not intersect a bounding volume containing a triangle, it cannot intersect the triangle itself. If a ray does intersect a bounding volume, all contained triangles must be tested for intersection with this ray. It is therefore beneficial to bound objects tightly, ensuring empty space within the volume is minimised.

The choice of bounding volume depends on the efficiency of the intersection test, the primitive(s) used in models, and the quality or tightness of the bounding provided. Spheres, ellipsoids and boxes are common choices for bounding volumes due to their relatively simple ray intersection tests.

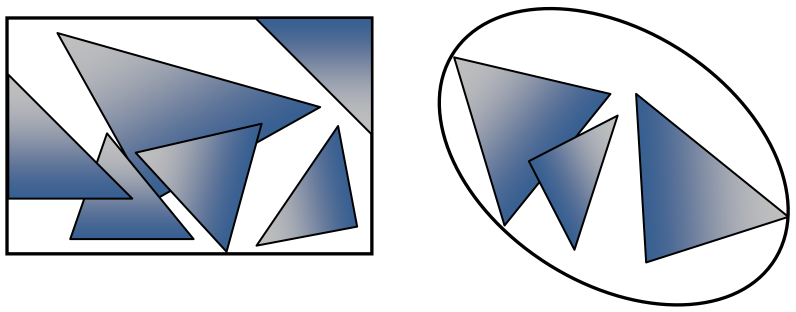


Figure 23: Bounding volumes illustration. 2D representation of primitives bounded in simple volumes of box (left) and ellipsoid (right).

4.2.2 *Bounding Volume Hierarchies*

The natural extension to bounding volumes is the bounding volume hierarchy. Multiple levels of bounding volumes are commonly used with the aim of further reducing the number of RTI tests.

To test a ray for model primitive intersections, it is first tested against each bounding volume in the highest level of the hierarchy. If no intersection occurs at this level, there can be no intersection with any triangle within the model. If an intersection does occur, then the hierarchy is recursively tested for each intersected volume. Primitives are tested for intersection if they are contained in an intersected volume.

Commonly used optimisations for BVH creation and traversal are discussed by Smits [44], Haines [45], Goldsmith and Salmon [46] and Arvo and Kirk [22]. Wald et al [47] discuss a change to the BVH algorithm making it more suitable to rendering dynamic scenes, where volumes are refitted to the objects they contain rather than rebuilding the complete hierarchy.

Recent works by Stich et al [20] and Popov et al [19] describe the use of spatial splitting for BVH construction. Stich et al use a SAH to control primitive splitting, resulting in more efficient hierarchies for traversal. Popov et al also use a surface area heuristic during BVH creation, and present a simple space partitioning based BVH creation algorithm.

4.2.3 *Grids*

Two popular grid partitioning schemes are octrees, in which the size of voxels (a volume in 3D space) can vary, and uniform grid partitioning [48] where voxels are of a fixed size.

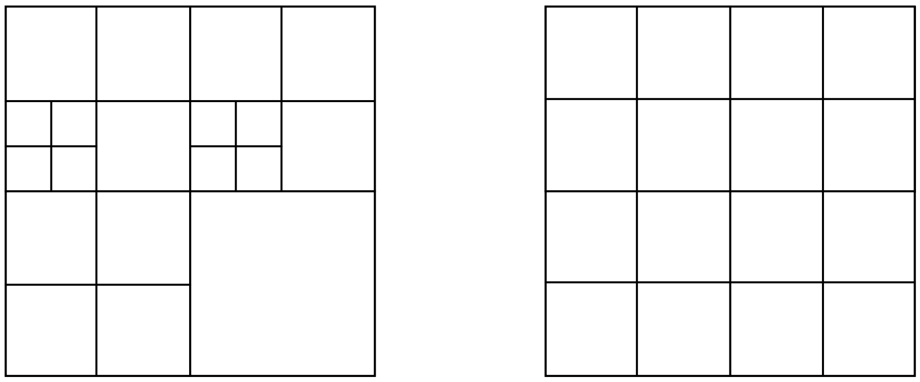


Figure 24: Grids illustration. Varying voxel size in octree (left), fixed voxel size in regular grid (right).

Grid traversal is performed as follows. First, the primitives residing in the voxel where a ray originates are examined. If the ray intersects any primitive in this voxel, the closest intersection has been found and no other voxels need be examined. If no intersection is found in the origin voxel, or there are no primitives within this voxel, neighbouring voxels are examined [49]. The primitives, if any, in these neighbouring voxels are then tested for intersection. This continues until a valid intersection is found or the partitioned space is exited.

Octrees are more difficult to traverse due to the varying size of their voxels, however this property can also be advantageous. Where there are many closely grouped primitives, these can be divided into several voxels, reducing the number of RTI tests to perform if a voxel is intersected. If there are no primitives in a large volume, then this can be left undivided resulting in fewer voxel examinations.

The commonly used example highlighting this advantage is a football stadium with a highly detailed teapot at the centre of the field. Storing this in a regular grid is problematic resulting in many sparsely filled voxels and one voxel that contains the many primitives forming the teapot. With octrees however, the teapot would be split into several voxels but the stadium would be stored in large voxels, each with few or no primitives contained.

In recent work, Lagae and Dutré [50] present two methods for representing and building a uniform grid. This is aimed at the rendering of dynamic scenes where the acceleration structure must be updated between the rendering of frames. Even though rendering can take longer when using a grid, rather than a *k*D-tree for example, the construction or update time is reduced. The premise of this work is that it is the sum of the time taken to perform both steps, construction and rendering, should be minimised.

4.2.4 BSP and kD-Trees

Binary trees reduce the number of RTI tests to perform by eliminating volumes within the model that must be considered as a ray is traversed through the tree. There are two main variants of binary space partitioning (BSP) trees, axis-aligned and polygon-aligned.

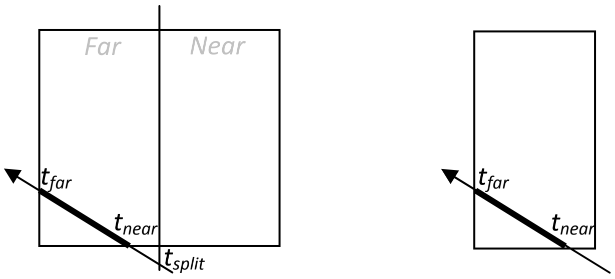
Polygon-aligned BSP trees [51] use the plane of a model element as the splitting entity that subdivides a spatial region into two parts. This makes models composed entirely of polygons or triangles particularly suited to construction as this form of tree.

The alternative form of the BSP tree uses a plane parallel to one of the coordinate axes as the splitting entity. This form of spatial subdivision is also referred to as a rectilinear BSP tree or an orthogonal BSP tree, but is commonly known as a *k*-dimensional binary tree [52]. This axis aligned tree is adaptable to the scene geometry due to the choice of axis and position for the splitting plane.

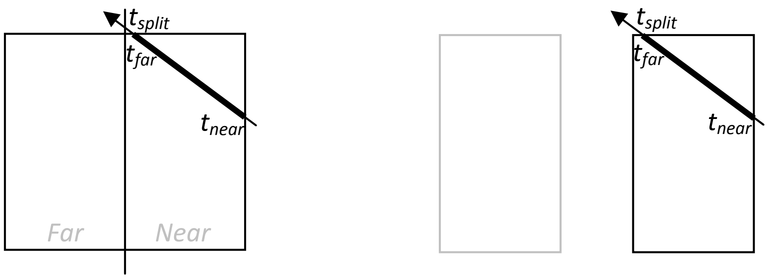
To test for intersections with model geometry, the path of the ray is followed through the tree, where the ray segment $[t_{near}, t_{far}]$ passing through the voxel represented at each node is calculated. This ray segment is first initialised to $[0, \infty]$, and is then clipped to the bounding box of the scene. This ray segment is updated as the ray passes through the different voxels in the tree. For each node traversed, the ray to splitting plane distance is calculated and then compared to the current ray segment. This leads to the three traversal options as detailed in Figure 25.

If a ray segment lies on one side of the splitting plane only, the sub-tree on the other side can be culled. If both sub-trees must be traversed, the ray-splitting plane intersection distance, t_{split}, is used to determine the order and the sub-trees are traversed in turn. The first child node is traversed using $[t_{near}, t_{split}]$ as the current ray segment, and the second using $[t_{split}, t_{far}]$.

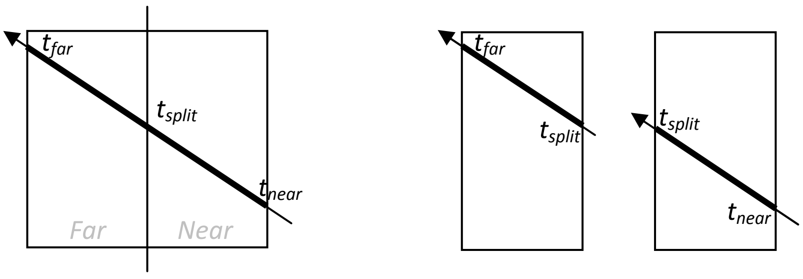
Using this front-to-back method of tree traversal allows early ray termination in many cases. If a valid intersection is found in a voxel, traversal can be halted immediately as subsequent voxels can only contain intersections behind this intersection. This leads to large performance gains in many cases for ray-based algorithms.



(a) Near side culled



(b) Far side culled



(c) Traverse both sides

Figure 25: Three traversal cases of a *k*D-tree. The ray segment is completely in front of splitting plane (a), the ray segment is completely behind the splitting plane (b), the ray segment straddles the splitting plane (c). Sub-trees investigated further are shown at right, with greyed areas culled.

There has been much previous work on generation of efficient trees and their traversal. Havran et al [53] present a tree-construction algorithm based on the statistical analysis of all possible traversal cases of a BSP tree, with the aim of reducing traversal time during rendering. Work from the same author [15] gives a detailed view of the costs associated with the construction and traversal of several acceleration structures, and in particular *k*D-trees. Szirmay-Kalos et al [23] also present a discussion of *k*D-trees in their comparison of acceleration structures. The quality of tree generated is dependant on the choice of axis and position at which to subdivide tree nodes, and several options for this are presented in the previously mentioned works. There are however alternative approaches, such as using a piecewise quadratic function to estimate the best splitting position based on few sample costs at varying split locations [18].

Woop et al introduce bounded *k*D-trees (B-*k*D) [54], which are data structures combining the advantages of bounding volume hierarchies with those of *k*D-trees. From bounding volumes, efficient support for dynamic scenes is gained, while maintaining the efficiency of *k*D-tree traversal. Instead of a splitting plane that divides space into two like a *k*D-tree, each node of a B-*k*D tree contains two pairs of axis aligned planes which bound the geometry of its two child nodes. As such, B-*k*D trees allow for simple and efficient updates when scene geometry changes, as only the bounding planes of the nodes are updated without altering the structure of the tree itself. Traversal of the B-*k*D tree is similar to that of a conventional *k*D-tree, in that the tree is traversed recursively using ray segments.

4.2.5 A Discussion on RAS Methods

We have presented several commonly used methods, all used to limit the number of RTI intersection tests that must be performed for any particular ray. Development of such acceleration techniques is an active research area, with new techniques and improvements to existing methods continually emerging.

From all techniques, *k*D-trees are the most commonly employed, with their use in renderers becoming a standard over recent years. With a well chosen splitting strategy and cost framework, *k*D-trees adapt well to varying scales and grouping of model geometry. In addition, the early exiting possible during traversal means that it is common that many fewer RTI intersection calculations must be performed than when other techniques, such as BVHs, are used. Previous work has stated that only three RTIs per ray must be computed when using a well built *k*D-tree.

For static scenes, a RAS can be generated prior to rendering. However for dynamic scenes, where the RAS must be continually rebuilt or updated between frames, this rebuilding time adds to overall rendering time. This time must be taken into account, although some recent techniques have showed that this build time can be significantly reduced, minimising any impact [18].

4.3 *Photon Map Search (PS)*

The photon map search function is specific to photon mapping, and as such, less work has been undertaken on optimising this function than the ray processing functions shared by other renderers. In this section we look at the common method used for photon map searching in software, optimisations, and other methods capable of performing the spatial point searching required.

4.3.1 Balanced kD-tree

The commonly used structure for a photon map is the balanced *k*D-tree [2]. In a photon map application, every node in the tree contains the location of a single photon, along with pointers to left and right sub-trees. Each node in the tree, except leaf nodes, partitions model space in one of the three coordinate axes using the position of the photon in the chosen axis as the splitting location. All photons in the left sub-tree are below this partition and all photons in the right sub-tree are above this partition, in the chosen axis. It is common to balance the photon map *k*D-tree in order to facilitate faster searching. For an unbalanced tree, the worst case complexity for locating a photon is *O(n)*, whereas for a balanced tree this reduces to a worst case of *O(log n)*.

To search the *k*D-tree, the distance from the splitting position of the *k*D-tree node to the search origin is calculated in the split axis. This distance is compared to the search radius and it is then decided whether the left, right, or both sub-trees must be searched, and in which order. Four traversal cases are shown in Figure 26.

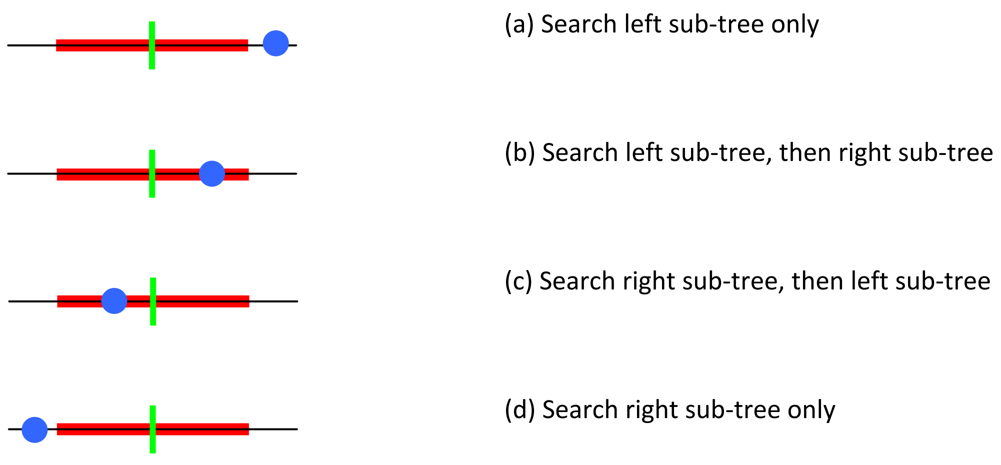

(a) Search left sub-tree only
(b) Search left sub-tree, then right sub-tree
(c) Search right sub-tree, then left sub-tree
(d) Search right sub-tree only

Figure 26: Four traversal cases of photon map *k*D-tree. Green line is the search origin in the split axis, red line is search volume in this axis, and blue circle is photon under test. Note that both sub-trees of the photon under test are traversed only if the photon is within the search volume in the split axis.

It is common for the *k*D-tree search to incorporate further testing to ensure that photons found are within a sphere, equal in radius to the photon search radius and centred at the search origin.

4.3.2 Alternative Spatial Point Searching Methods

Other options which could be used for spatial photon searching include locality sensitive hashing (LSH) [55, 56] and the *R*-tree [57] data structure. In LSH, similar items are mapped to the same hash buckets. During a search, only photons within buckets identified by the hash of the search origin need be tested. These photons would be spatially close to the search origin.

R-trees split space using minimum bounding rectangles (MBR), with each node of an *R*-tree having a variable number of MBRs which can overlap. During a search, the MBRs are tested to see whether child nodes should also be searched, meaning that it is possible for many nodes to be eliminated from the search.

Another option uses a reorganised *k*D-tree, using lessons learned from the construction of high quality BSP trees for use as RAS [58]. A cost function is used, however rather than using the surface area heuristic as used for the construction of RAS, the cost is based on a voxel volume heuristic. This approach has shown that a maximum of 3.4 times speedup in search time is achievable, at the expense of increased complexity and possibly longer tree build time than the traditional balanced *k*D-tree approach. This technique has not been widely adopted.

4.3.3 k Nearest Neighbours

A *k* nearest neighbours search finds the *k* nearest neighbours to a query point. In the case of photon search, for an area where photons are sparse, a large area would be covered by the *k*NN. This large area is used in the photon density calculation, meaning noise in the sample is minimised. Conversely, if there is a dense area of photons, the area covered by the *k*NN would be smaller. Again this small area would be used in the density calculation, meaning that the blurring of the edges of any focused light will be minimised.

As this approach adapts to local variations in photon density, it is very useful in the rendering of optical effects such as caustics, in which photons are often concentrated in small regions.

In practice for a *k*D-tree *k*NN search, a max heap is often used to hold the *k* nearest neighbours to the search origin. When the heap is full, the distance from the search origin to the furthest photon in the heap is known, and the search radius is effectively modified in order that *k*D-tree branches containing only photons further away than this can be omitted.

4.3.4 Irradiance Caching

To improve the performance of the irradiance calculation, irradiance caching can be used. This is a method for caching and reusing irradiance values [2]. As indirect illumination often changes slowly in a scene, it is a good candidate for reuse via interpolation.

In this approach, the irradiance is only computed at selected locations on surfaces in the model, and the irradiance values required for other locations are found by interpolating between these existing values. The decision of whether to interpolate or calculate the irradiance for any location depends on the previously computed values. In this decision, a weight is calculated for the existing samples, the value of which indicates how good the interpolated estimate of irradiance is for this location. If no previously computed irradiance value with sufficient weight exists, a new one is calculated for the location.

In practice, it is expensive to keep computing the weight of all previous irradiance values repeatedly. However, as each irradiance value is useful in only a small area of the model, a sphere in which any irradiance value is useful can be formed. This radius is based on the distance that the weight becomes too low to be useful. This sphere can be stored in an octree for efficient querying of whether previously computed irradiance values can be of use at a particular location.

Another form of irradiance caching is introduced by Christansen [59]. In this approach, again irradiance values are pre-computed, but in this case at photon positions, or at a subset of these positions to minimise computation. Photon locations are a good choice as photons are inherently dense in locations of large illumination variation.

The pre-computed irradiance values, along with the surface normals, can be stored along with the other data for each photon in the photon map. To recall an irradiance value during rendering, the irradiance value stored with the nearest photon which has a similar surface normal is used.

4.3.5 A Discussion on PS Methods

By far the most commonly used photon map data structure is the balanced *k*D-tree. This structure is used in the traditional photon mapping algorithm due to the efficiencies in spatial point range searching offered. This is the technique used in the photon map renderer designed as part of this work.

There have been several attempts to reduce photon search time using methods such as irradiance caching and interpolation; however such approaches could have an impact of degrading the realism of generated images. A recent proposal which would not decrease image realism simply reorganises the *k*D-tree based on a cost model, such as those used in RAS *k*D-tree generation [58].

There are of course other approaches, many of which are used in other fields; however none have been widely adopted for use as photon map data structures.

5 OVERVIEW OF EXISTING HARDWARE

In the previous sections we have identified three core rendering functions of a photon map renderer and looked at common methods that are used within these functions. In this section we look at any existing hardware created for the acceleration of these methods.

Two of the core functions, those dealing with ray-triangle intersection processing, are shared by various other ray based renderers including ray-tracing. One of the aims of the recent past was the generation of ray-traced images in real-time, and as such there has already been a great deal of research into the acceleration of these functions. The photon map search function used in the photon mapping algorithm is a recent addition to ray based rendering with less hardware available for its acceleration.

An overview of various commercial and research hardware designs for acceleration of the RTI, RAS and PS functions is provided in the following sections.

5.1 Existing Ray-Triangle Intersection Acceleration Hardware

In this section we look at a range of both commercial and research hardware for the acceleration of ray-triangle intersection processing.

5.1.1 Custom Hardware Based RTI Designs

Advanced Rendering Technologies Ltd detail the use of a ray processor unit based on 20-bit logarithmic number system (LNS) arithmetic, to be used in a parallel formation for the generation of ray traced images [60]. Each ray processing unit is used to calculate the required output for a group of pixels it is allocated, with multiple parallel units working simultaneously on separate rays. Each unit determines all intersections for its current ray with bounding volumes and model object surfaces. These model elements are stored in a tree structure and are presented to the ray processing units as the tree is traversed. The coordinates of the closest intersection with the bounding volume or object surface as well as the optical characteristics of this surface are stored in intersection memory. This data is used by the output generation unit where pixel intensities and colours are calculated.

There is little performance data presented for this system. Example execution times for ray generation and ray redirection units are shown, but there are no figures presented for ray-object intersections. Only an example ray-object intersection unit is illustrated, showing only a ray-plane intersection calculation performed in a six stage pipeline.

The following possible issues were identified with this design:

- The arithmetic format used means that logarithmic values in the range -32.0000 to +31.99988 can be represented, equating to real values in the range -4x10^{-9} to 4x10^{9} to an accuracy of 5 decimal places. Although no artefacting was visible in images produced with this setup, performing RTI calculations with a greater range and accuracy may be preferable in the most realistic renderings. This would reduce the possibility of the accumulation of any error, which could otherwise lead to visible artefacting.
- To maintain the expected performance, all rays are initialised in ray memory before rendering is commenced meaning that all units can be continuously supplied with data. This would be difficult with photon mapping, where rays tend to exhibit randomness and are serially dependant.

A series of special-purpose products for RTI acceleration has until recently been commercially available from Advanced Rendering Technologies (ART-VPS). These were based on the AR250 [61], AR350 [3] and AR500 [62] ray processors. These processors use special-purpose pipelines for calculating ray-triangle intersection calculations. The AR250 makes use of 32 32-bit floating point arithmetic units and operates at 50MHz. Each of these devices requires 106mm^2 of silicon in the 0.35µm process used. These units were used in the RenderDrive network appliance. The AR350 was progression of the AR250 and contains 64 32-bit floating point arithmetic units consuming 110mm^2 of silicon in a 0.22µm process. This was also used in the RenderDrive, as well as the PC plug-in PURE card. The last generation of the ray processor, the AR500, is used in the RayBox device. Each RayBox contains 14 dual core ray processors and interfaces to a PC via PCI Express.

There is limited performance data available for these devices. Overall rendering time of a single model is shown for a RenderDrive containing 16 AR250 chips hosted by an Intel Pentium 2 based PC. This shows a 98% reduction in total rendering time when the RenderDrive is used, however no details are available for the model or rendering setup which would allow us to calculate accurate ray-triangle intersection performance.

Recently ART-VPS have dropped all rendering acceleration hardware from their catalogue, and now offer their ShaderLight software product which allows artists to view renderings of their models throughout development. It is our suspicion that the ray processors were unable to maintain a sufficient margin in render-time reduction over PCs, especially with the fairly recent trend for multiple core devices. Compounding this, ray intersection processing accounts for a reduced proportion of execution time when more realistic rendering algorithms are used, such as photon mapping. No hardware for accelerating the additional functions of more realistic renderers was made available.

‘SaarCOR’ is a more comprehensive solution to the entire ray-tracing procedure, which uses *k*D-trees as a RAS and processes packets of rays [41, 63-65]. All rays of a packet are traversed through the tree simultaneously, with a node inspected if any ray is found to intersect. A bit-vector associated with a packet of rays is used to keep track of ray activity in the current tree branch, allowing only active rays to be processed further. The ray tracing core of the design is used to perform this ray packet traversal through the *k*D-tree. Tree traversal is dependent on ray coherence, as packets of rays are effectively treated as a single ray, reducing the amount of data that must be fetched from memory.

The intersection unit of the ray tracing core is used to test rays for intersections with triangles in the leaf nodes of the *k*D-tree which are identified during traversal. The ray-triangle intersection test employed is based on Arenbergs algorithm [40]. The intersection unit requires ray data that has been pre-processed using affine transformations into unit triangle coordinate space. ‘SaarCOR’ provides a dedicated transformation unit for this purpose, in which all rays of a packet are transformed sequentially. For packets of *n* coherent rays, the shared ray origin need only be transformed once, resulting in *n+1* transformations. However, if the packet of rays are not coherent, *2n* transformations must be performed for both the origin and direction of each ray. The authors state that, in ray-tracing, most packets of rays can be considered coherent as reflected rays retain their shared origin if they are reflected or refracted by one planar surface, as shown in Figure 27.

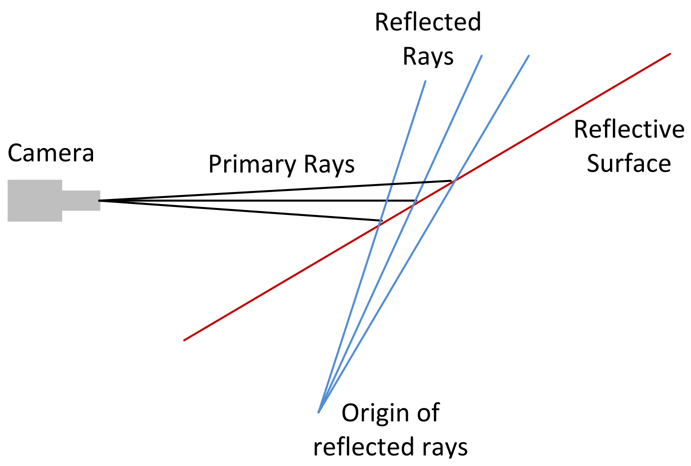


Figure 27: Rays reflected and refracted by a simply modelled surface can be found to share their origin.

After ray transformation, intersection with the triangle becomes trivial but still makes use of a division operation to calculate ray-plane intersection distance. The unit performing this calculation consists of a simple pipeline containing 1 divider, 3 adders, 2 multipliers and 3 comparators. Due to the required divider, the intersection unit consumes over one fifth of total logic gate count for the prototype based on the use of 32 rays per packet. If including the 9 adders and 9 multipliers of the required transformation unit, then this rises to more than one half of total logic gates.

A simulated 'SaarCOR' was able to render several models, ranging in complexity from 34k to 3.6M triangles, at 4 to 113 frames per second (fps) using between 1 and 4 ray tracing cores operating in parallel. The number of ray-triangle intersection calculations actually performed is not known. Details of an FPGA based prototype implementation, using 24 bit floating point arithmetic and operating at 85-92MHz is also provided [64]. Results for this implementation show that low resolution renderings (512x384 pixels) can be performed at up to 60 FPS for a range of models varying in complexity from 800 to 1.8M triangles. Detailed statistics for the prototype show that a maximum of 68M ray transformations can be completed per second, and consequently the same number of triangle intersection calculations can be done in this time. However, these figures are based on the assumption that all packets of rays are coherent, which is unlikely to always be the case. No further details were provided on this matter.

The following possible issues were identified with this design:

- An interesting problem noted by Schmittler highlights a drawback to the use of Arenbergs RTI method [41]. Transformation of ray and primitive from world space to unit triangle space can lead to a loss of precision (twice, for triangle and ray) due to large differences in scale. This caused noticeable artefacting in renderings produced using 'SaarCOR'.
- Ray coherence may not always be as abundant as stated. Rays generated in photon mapping tend to be more random in their nature and serially dependant. One cause of this is the more accurate representation of model surfaces, such as when microfaceting is employed. Here, rays will be reflected or refracted randomly within a range dependant on the microscopic roughness of the surface.

'RPU' [66, 67] moves away from the fixed function units of 'SaarCOR', and offers a more programmable ray tracing hardware architecture. The core shader processing unit (SPU) is used to perform triangle intersection calculations and shader operations, using four component single precision floating point or integer vectors as the basic data type. A multithreaded approach is used to take advantage of data parallelism. For every primary ray a new asynchronously processed thread is started, the state of which is maintained in hardware. The execution of a SPU then switches between threads as required. To exploit data coherence, chunks of threads are executed synchronously in single instruction multiple data (SIMD) mode in parallel by multiple SPUs. The number of SPUs in a chunk is fixed for a particular implementation, and depends on the expected coherence within a dataset to be processed synchronously by a chunk.

Traversal of chunks of rays through a tree is performed using a fixed function *k*D-tree (or B*k*D-tree for dynamic scenes) traversal unit, with one provided per chunk thread. Triangle intersection calculations are performed in the SPUs, using a test based on Arenbergs algorithm [40].

'RPU' has been prototyped using FPGA devices and is shown to offer up to 20 FPS renderings of models containing 800 to 1.8M triangles at 512x384 pixel resolution. The FPGA was clocked at 66MHz. Estimated performance results for two 130nm application specific integrated circuit (ASIC) implementations and a 90nm implementation show that performance over the prototype can be significantly improved [68]. Projections for the 130nm integrated circuit (IC) containing a single instance of the RPU design show that 6 to 24 FPS could be achieved for low complexity models (up to 85k triangles) rendered at 1024x768 pixel resolution with the IC clocked at 266MHz. The performance is reduced in comparison to 'SaarCOR', which is noted to be due to the overheads imposed by additional programmability.

The following possible issues were identified with this design:

- The 'RPU' prototype, like 'SaarCOR', uses a 24-bit floating-point representation. It is however noted that in an ASIC implementation a 32-bit wordlength would be used, suggesting the authors envisage that a greater range and precision is needed.
- The efficiency of the processing remains dependent on ray coherence, which is reduced in photon mapping, as described in reference to 'SaarCOR' earlier.
- 'RPU' is scalable, meaning the number of units used to perform a rendering can vary, but this scalability is limited. Multiple units can be used provided that multiple accesses to a common model description in memory can maintain the required throughput.

The '3DCGiRAM' [69, 70] consists of multiple processing modules, each working on separate subspaces of the 'SEADS' regular grid RAS used. Each module is responsible for calculating intersections with triangles stored in the grid subspaces it is allocated. A ray distributor unit identifies which module contains the subspaces first to be intersected by a ray and forwards the ray to this module. When a ray is received by a module, the 3D Digital Differential Analyzer unit specifies one intersecting subspace contained where ray-object intersection calculations should be performed. This subspace and the ray data are sent to one of the intersection calculation units within the module for processing. If no intersection is found, the 3D Digital Differential Analyzer specifies the next subspace to be tested. If an intersection does occur with model geometry, a secondary ray is generated by another processing unit and is sent to the 3D Digital Differential Analyzer.

The intersection calculation unit is based on Arenbergs algorithm [40]. Each intersection unit takes 32 cycles to process one ray-triangle intersection calculation, which includes transforming the ray to unit triangle space. The initial paper states that an intersection calculation unit is composed of one 3-stage pipelined fixed-point multiply-adder, while a divider is shared between two intersection units [69]. However, the subsequent paper suggests floating-point arithmetic will be used [70]. There is no detail given on the exact number format in either case.

As a single ray-triangle intersection takes 32 clock cycles to complete, a relative performance of 1 RTI per clock cycle is indicated for a '3DCGiRAM' containing 32 units. This configuration would rely on the use of 16 shared dividers, and be dependant on one (of the paired) intersection units stalling during the first 16 cycles of an intersection calculation, so there is no conflict for the divider resource. Another possible issue is that performance relies on a pool of ready-to-evaluate rays, intersecting different subspace groups, to keep all the modules busy. The indicated performance of 1 RTI per clock cycle is dependant on the intersection units being consistently supplied with data. This proved difficult during the evaluation of a prototype ASIC design due to the very large memory bandwidth required. This was based on a 3DCGiRAM running at 333MHz containing 32 intersection units, for renderings at 10 fps.

A recent proposal by Kim et al [71] is able to perform ray-object intersections in the context of collision detection. The design consists of a set of pipelines for detecting intersections between a variety of geometric objects, the RTI pipe being an implementation of the Möller-Trumbore algorithm. This was prototyped using an FPGA device. Interestingly, and unlike other proposals, the FPGA did not make use of an RAS due to resource constraints, and thus provides a baseline for comparison with our own RTI device which can also operate in the absence of a RAS technique.

The performance of the FPGA was compared with an equivalent software implementation running under OpenGL and Windows XP, on a 2.8GHz Xeon CPU. Over several frames, the CPU maintained a processing rate of 259,572 RTIs in about 2,100ms, i.e. 8.09µs per RTI. Note that this figure is comparable with the RTI rate achieved by our own software. The Xilinx Virtex2 P70 FPGA delivered the same processing in 31ms, or 119ns per RTI, seventy times better than the CPU. The experiment was repeated using an NVIDIA GeForce 7800GT and the Cg shading language. This required about 470ns per RTI.

Like the previous proposal, this design is also directed at the simultaneous processing of multiple pre-initialised rays in memory. The performance indicated relies on having this ray data available in memory.

Hanika and Keller [72] propose the use of fixed point arithmetic in a ray processing unit. A fixed point representation is used to reduce the silicon area required for the arithmetic elements needed to perform the ray computation over a floating-point based equivalent. Previous attempts to use fixed point arithmetic in this field has led to artefacting in the images created due to the limited range and precision available. In this implementation, precision was enhanced by examining the ranges of values used and exploiting the equidistant spacing between fixed point numbers. The fixed point processing elements are used in a pipelined version of Badouels algorithm, which was used to perform the ray-triangle intersection tests.

No performance data is presented, however it is stated that images have been created using a software simulator based on these techniques, while an FPGA based implementation is in development.

5.1.2 GPU Based RTI Designs

Carr et al [73] made an early proposal for executing RTI operations on a GPU, with the remainder of the ray-tracing algorithm being executed on a CPU. The ray engine works on caches of coherent rays, and uses a pixel shader program performing the Möller-Trumbore method to test for ray-triangle intersections.

In terms of performance, the Radeon 8500 based implementation was able to perform 114M ray-triangle intersection tests per second. Simulations, based on a GeForce 4 implementation, show that 114k-207k rays can be processed per second in the rendering of some simple models consisting of between 2.6K and 34K triangles. This included the processing of the RTI acceleration structures employed.

Significant artefacting was present when this method was implemented on the ATI Radeon 8500 due to its 16-bit fixed point based pixel shader, however it was noted that this was cured in simulations of more precise GPUs.

Purcell et al [74] propose an implementation of a ray-tracer on a hardware platform not available at the time, but based on likely future hardware. The authors show how ray-tracing can be mapped to a stream processing model, and also how the performance of GPU stream based ray caster is competitive with the equivalent function executed on a CPU. For a GPU running at the same operating clock rate as a GeForce 3, 56M RTIs/s could be computed, increasing with the number of GPU instructions that can be executed per second on future GPUs.

5.1.3 RAS Acceleration Hardware

Is there a need for hardware RAS techniques if the execution of RTI calculations can be accelerated sufficiently? A simple answer is that, although some proposals have shown that some acceleration can be achieved without the use of a RAS, there are no offerings to date that have been able to offer adequate acceleration of RTI calculations to negate the need for some form of RAS.

Without RAS techniques, the number of RTI calculations to perform would be very large indeed. When combined with the fact that all RTI methods contain some complex processing, we can see the reason behind most existing ray-tracing acceleration designs incorporating some form of RAS. In this section, we look at some of the RAS strategies used by the RTI hardware previously described.

In the ART LNS based design, the intersection unit is able to handle intersections with both model geometry and bounding volumes. The bounding spheres used are stored in a tree structure for ease of traversal and broadcasting by the main processor. While this approach can reduce the number of RTIs to perform, it is by no means the most efficient method as spheres tend to have an abundance of empty space around the objects they enclose, especially the case for objects such as polygons or triangles.

The 'SaarCOR' and 'RPU' designs make use of *k*D-trees to reduce the number of RTIs. In 'SaarCOR', a dedicated traversal unit takes packets of rays from a ray generation unit, and traverses through the tree with them until a node containing triangles is found. The intersection unit then works with the list of triangles in this node.

In a similar approach to that of 'SaarCOR', custom traversal processing units are provided in 'RPU'. These units traverse packets of coherent rays through the *k*D-tree. Rays in these packets are deactivated as necessary during traversal through use of an associated bit vector. If a non-empty leaf node is found, the contents of this node are tested for intersections. For both systems, little performance data for *k*D-tree traversal is presented.

In a version of 'RPU' modified to deal with dynamic scenes, B*k*D-trees are used as a spatial index structure [68]. These trees are updated by a dedicated update processor working in a bottom up approach when a scene changes. The trees are traversed by a dedicated traversal processor, which uses multiple fixed function traversal units in a SIMD approach. Little performance data is presented, except suggestions that traversal typically requires between 50 and 100 steps, and updating the tree for a range of models composed of up to 85k triangles can take up to 602k cycles.

Mahovsky [75] presents an axis-aligned bounding box (AABB)-ray overlap method for use in BVHs. Integer representation is used to facilitate a reduction in the memory required to store AABB coordinates, while integer based ray-AABB overlap calculation hardware is also derived and compared to the floating-point equivalent. This work concludes that the integer test requires only half of the equivalent floating-point circuit area.

The '3DCGiRAM' uses the 'SEADS', or Spatially Enumerated Auxiliary Data Structure for a RAS. This data structure is independent of model geometry, and can be considered as a 3D raster grid. It is stated that a 3D Digital Differential Analyser is present within each module, which specifies one of the intersecting grid subspaces for ray-triangle intersections to take place. This unit is a fast line generator which uses ray data to find intersected subspaces in the grid. A maximum of 7 cycles are taken to test whether a subspace is intersected, 6 of which are used to recall data about the subspace from memory.

In the work by Hanika and Keller, bounding interval hierarchies [76] are used as an acceleration structure. Similar to B*k*D-trees, two planes parallel to one of the co-ordinate axes are stored per node, along with pointers to two child nodes. Like the RTI unit proposed by the authors, fixed-point arithmetic is again used in the processing of the acceleration structure. It is noted that care has to be taken in the intersection of the ray with the axis-aligned planes, ensuring there is sufficient precision. The design was simulated in software, however no performance data is provided.

In the majority of existing RTI acceleration hardware, the use of RAS has been required to reduce the number of RTI tests to perform and increase the acceleration achieved. However, the work by Kim et al [71] showed that some acceleration could be achieved without the use of such a technique.

5.1.4 Justification for new RTI Acceleration Hardware

Regarding existing designs, the following observations are relevant:

- Some are based on GPUs which have a finite limit on their performance and available resource, restricting the scalability of any design. Constructing a large array of GPU devices to overcome this limit would be cumbersome, and has not been suggested in any of the proposals.
- Apart from the ART systems, none appear to be efficiently scalable without any practical limit, and so suited to a massively parallel arrangement that could offer further increased acceleration. This is not to say that existing solutions are not scalable, as some have shown to be. However with simpler hardware combined with an alternative processing model, e.g. brute-force versus smart, there may be the opportunity for increased scalability.
- Some rely on ray coherence, which is abundant in ray tracing but not in photon mapping which is characterised by random rays that are serially dependent. Several proposals also rely on a pool of ready-to-evaluate rays in memory, which are difficult to calculate in photon mapping for the same reasons.
- Several make use of Arenbergs RTI method, which was shown to have caused artefacting in images produced using the 'SaarCOR' prototype due to the loss of accuracy in the transformation of the ray and triangle.
- Some use fixed point or short wordlength floating point or logarithmic representation in their arithmetic. For the most realistic of renderings, it may be preferable to use a representation of increased range and precision to minimise the accumulation of error in calculations.
- Most employ some form of acceleration structure to reduce the number of RTI calculations to perform. This takes time to compute, although this is not quantified in any of the proposals.
- None have been considered in the context of photon mapping, which requires integration with photon search hardware.

Due to the diminished amount of ray coherence in photon mapping compared to ray tracing, the efficiency of execution of any new design for processing RTIs should not be dependent on this attribute. For photon mapping applications, we see most benefit in the use of a scalar unit. Such a unit would be able to process ray-triangle intersection calculations, for a single ray, as quickly as possible. As the same ray is often tested against many different triangles, parallel execution of these tests would make sense.

Another property of any new system should therefore be improved scalability. While existing solutions are scalable to some extent, their use of large hardware units and complex data structures prohibit large scale replication.

5.2 Existing Photon Map Search Acceleration Hardware

Several workers have proposed hardware solutions for the acceleration of the photon map search. There are several GPU based proposals, but very few custom hardware designs. Some of the most applicable works are summarised in this section.

5.2.1 Custom Hardware Based PS Designs

Proposals for acceleration of the PS function in custom hardware devices are few and far between. We are only aware of a single FPGA based prototype implementation based on such a design.

Singh and Faloutsos describe pipelined hardware capable of *k*D-tree traversal applicable to photon map searching [77, 78]. This hardware is based on the idea of reverse photon mapping, where instead of searching for photons around query points, query points are sought about photon locations. In this work, the reverse photon mapping algorithm is modified so that each photon contributes directly to pixels in the final image being generated. A difference between traditional and reverse photon mapping is in the density estimation, as sample-point estimation is used in place of a *k*NN search.

To generate an image, two ray tracing units are used, one for camera rays and the other for photons. Intersection points with the camera rays, including secondary camera rays, are stored in a *k*D-tree. Photons are traversed through this tree, where camera points affected by these photons are found. The hardware described contains several tree traversal units, which operate in parallel for distinct photons. During image creation, the hardware shader takes photon-camera point pairs and computes the partial sum which is accumulated in the image.

No timing data is provided for this design, however the number of shader operations and the required memory bandwidth for some fairly complex models rendered at 1 FPS are shown. These models have between 3.7M and 23.6M camera points. It was noted that these figures fall within the limits of performance for current graphics hardware.

Heinzle et al describe a hardware *k*D-tree search architecture for spatial point searching [79]. This hardware is capable of performing both *k*NN and Euclidean neighbours (εN) searches, which is defined as the set of neighbours within a given radius of the query point. As well as the configurable *k*D-tree traversal core, the architecture also incorporates a caching mechanism to exploit spatial coherence between queries. The *k*D-tree traversal unit is composed of a node stack, a stack recursion module and a node traversal unit, and can accommodate up to 16 parallel threads.

This design was implemented using both FPGA devices and an NVIDIA GeForce 8800 GTS GPU. It was noted that for *k*NN searches, the FPGA based implementation clocked at 75MHz had 68% of the performance of a modern 2.2GHz dual core CPU.

5.2.2 GPU Based PS Designs

Proposals for acceleration of the PS function have largely been based on modifying the algorithm to make it suitable for execution on a GPU.

Purcell [80] implemented the photon search algorithm on a programmable GPU. Instead of using a *k*D-tree as the data structure for the map, a uniform grid was used which was held on a NVIDIA GeForce FX5900 Ultra. The grid is searched by incrementally expanding the search radius from the search origin, examining sets of grid cells concentrically about the query point. Performance data shows that computing the radiance estimate was the most costly operation for several test renderings, which did not exhibit the same illumination quality as a software reference rendering.

The actual radiance estimate timings ranged from 4.6 to 52.4 seconds for three test models, rendered at low resolution (512x384 pixels and 512x512 pixels). These timings were based on the use of between 31250 and 62500 grid cells with between 5000 and 65000 photons stored, and photon lookup sizes of 32, 64 and 500. No timing comparison was made to a software renderer.

Czuczor et al [81] use textures to store photon locations, replacing the *k*D-tree searches in traditional photon mapping with texture filtering. As photon hits are stored in texture space, searches are also performed in texture space, which means looking for photons in a neighbourhood around a texel (texture element). For points that are close on a surface, it is often the case that their respective texels will be close, thus these texels can be included in a search. However, this is not always the case, as texels can be close even for disparate object locations. The authors alleviate this problem by storing surface identities in the texture alongside the photon hits.

The reflected radiance computation is implemented as a pixel shader routine on a NVIDIA GeForce 6800 GT. Texels in a square locale around the query point are found, and a decision is made whether they should be used in the search neighbourhood. For texels within the neighbourhood, their photon power is multiplied with the bidirectional reflectance distribution function (BRDF) [82] of the point of interest, which describes how much light is reflected depending on the light and viewer position relative to the surface normal and tangent, and the result is added to the reflected radiance at this point. Performance data for this work revealed that two simple models could be rendered at up to 45fps.

A problem was noted with this approach in that more than one photon hit cannot be stored in a texel for specular surfaces. This is due to the direction dependence of the photon reflection. The approach taken in this work to alleviate this issue was to use higher resolution textures and to purge photon hits for dense regions. When hits are purged, the authors state that the power of existing photon hits should be scaled appropriately. Having limits on the number of photons which can be stored in particular areas of illumination could be a barrier to the use of this method for very realistic renderings.

Ma and McCool [83] present a hashing solution to the problem of approximate *k*NN point searching (A*k*NN) which is applicable to photon mapping. Hash functions are used to categorise photons by their spatial positions. Spatial areas are preserved, meaning that photons close in domain space are also close in hash space. Thus, photons within the same hash bucket as the query point can be assumed to be close to the query point, and are good candidates for a *k*NN search. This technique has the useful property that multi-level memory lookups are not required, avoiding operations serially dependant on one another, such as in a *k*D-tree search.

Photons are organised into fixed-size memory blocks, which are then in turn inserted into hash tables. Each hash bucket in the tables can store multiple blocks of photons. A *k*NN query is submitted to all hash tables, which reveal all candidate blocks in a single hash container for each table which match the query. Depending on the location of the thresholds within the hash table, different hash buckets will map to varying sized volumes in domain space. This could mean that a single hash bucket could represent only a subset of the domain volume and photons sought. To mitigate this problem, multiple hash tables are used, each with different thresholds, which makes the retrieval of a more complete neighbourhood around the query point possible.

The final A*k*NN photon set comes from scanning the candidate set for the nearest neighbours. To increase the speed of the query, while having a possible detrimental effect on accuracy, it is possible to use only a subset of the returned hash containers. For highly realistic renderings, it is likely that such a technique would not be employed as noise could be introduced if a statistically unrepresentative set of photons is used in illumination calculations.

Few details of a hardware implementation were provided, although it is stated that this hashing approach could be suitable for implementation using future programmable graphics hardware. No performance data was provided.

McGuire and Luebke present the image space photon mapping algorithm which is processed using both the CPU and GPU [84]. In this proposal, expensive photon mapping operations of tracing the photons from the emitter to their first intersections, and calculating the irradiance estimates using the density of local photons are reworked as image-space operations which can be performed using rasterisation on the GPU. First, a bounce map of emitted photons is formed on the GPU. This is used as photons are traced conventionally on the CPU before photon volumes, based on the probability density of each photon path, are rasterised on the GPU scattering indirect illumination.

Performance results show that up to 39 FPS renderings for models ranging in complexity of up to 1.2M polygons are achievable. However, artefacting was noted in some situations.

5.2.3 Justification for a new Photon Search Unit

The following observations regarding existing photon search acceleration hardware are relevant:

- Many solutions try to rework the photon search to another form, where the required calculation can be performed more quickly and easily. In many cases this introduced artefacting, making such techniques unsuitable for the synthesis of highly realistic graphics.
- The GPU based proposals limit scalability due to the performance and resources available, much the same as RTI proposals for GPU implementation.
- None are scalable into a massively parallel arrangement. GPUs don't have sufficient resource and the size of the custom hardware structures described prohibit large scale replication.
- Integration with a ray processing unit has not been considered.

All existing proposals lack scalability, which is an obvious solution for photon search acceleration, with the many stored photons all being subjected to the same test.

Some proposals trade off accuracy for speed, using only a subset of photons within search boundaries, increasing the possibility of artefacting or noise being introduced into an image. Such an approach would not be possible for the synthesis of highly realistic images where visible errors cannot be tolerated.

6 ACCELERATION HARDWARE DESIGN

In previous sections we have confirmed that photon mapping rendering time is shaped by three core functions, all of which would be suitable for acceleration using custom hardware structures. These are:

- Processing RTI calculations
- Processing the RAS technique employed
- Performing photon map searches

In this section, we present hardware designs for acceleration of two of the three core functions identified: RTI processing and photon map searching.

6.1 RTI Acceleration Hardware Design

In this section we present a design capable of accelerating the processing of RTI calculations.

The major advantage of this design is its scalability. Multiple units can be used to process all RTIs for a single ray or multiple different rays in parallel. Apart from the additional hardware requirement, there is little extra cost associated. Reduction in RTI execution time scales almost linearly with the number of units used.

Another advantage is the non-dependence on ray-coherence for processing efficiency. The design is for a scalar unit, which works with a single ray and determines the closest triangle to the ray origin intersected from a number tested, and does so as quickly as possible. This makes the unit suitable for accelerating RTI calculations in photon mapping, where rays do not show the level of coherence of those in ray-tracing. The performance of many existing hardware proposals is dependant on processing coherent rays.

We begin this section by looking at the various design decisions made, such as the choice of RTI method and arithmetic format, before looking in depth at the structure and operation of the RTI acceleration unit.

6.1.1 Design Options

The three main options which must be considered in this design are:

- Hardware structure
- RTI method
- Arithmetic system

As the design is for the acceleration of RTI processing, execution speed is the main driver behind our design decisions.

6.1.1.1 Choice of Hardware Structure

There are two obvious options in terms of hardware structure. The first would be to use a microcode based processor to execute the chosen RTI method. The advantage of this method is the simplicity of the hardware required. Arithmetic elements can be reused in such a design, meaning the silicon area requirement is minimised. The major disadvantage is the speed at which a RTI calculation can be performed. It is unlikely that, even using the most efficient RTI algorithm, sufficient acceleration could be provided over the equivalent RTI test in software executed on a PC.

The second option is to use a pipeline structure. In such a formation, a separate arithmetic element must be provided for every operation performed in the RTI method. As arithmetic elements are utilised in every clock cycle there is no possibility of reuse. This has an impact on the silicon area requirement, which will be larger than a microcode based unit for any RTI algorithm used. The major advantage however is the execution speed achievable. In a pipeline, input is supplied on every clock cycle, and a result is output on every clock cycle after a period of latency. This means that a single RTI acceleration unit based on a pipeline structure could perform 1 RTI test per clock cycle.

If the size of such a pipeline could be minimised, this would allow multiple of these pipelines to be used in a parallel formation, performing multiple RTI calculations per clock cycle.

6.1.1.1.1 Brute Force Approach

The above approach of using multiple pipelines could be extended further, into a massively parallel arrangement, depending on pipeline size. As, in such an arrangement, multiple RTI calculations could be performed simultaneously, this raises the suggestion that an alternative processing strategy could be used.

In software, a RAS is often used to reduce the number of RTI calculations that must be performed in a smart approach. However, if many RTI calculations can be performed in simultaneously in a massively parallel approach, it may be possible to eliminate the use of a RAS. This would lead to the elimination of both the RAS construction prior to rendering, and its traversal during rendering.

In such a brute force approach, each ray would need to be tested for intersection with every triangle in the model being rendered. This would lead to many more RTI calculations being performed than in the afore mentioned traditional smart approach, as shown in Table 10.

Number of RTI Calculations	**Model 1**	**Model 2**	**Model 3**	**Model 4**
Smart Approach (*k*D-tree RAS)	$7.985\text{x}10^{7}$	$9.753\text{x}10^{7}$	$1.293\text{x}10^{8}$	$1.317\text{x}10^{8}$
Brute Force Approach	$1.596\text{x}10^{11}$	$3.349\text{x}10^{11}$	$5.245\text{x}10^{11}$	$7.627\text{x}10^{11}$
Times Increase	***1999.445***	***3433.363***	***4055.946***	***5793.562***

Table 10: Number of RTI calculations performed in Smart and Brute Force approaches.

To achieve acceleration while performing the significantly increased number of calculations emphasises the need for an efficient RTI intersection unit, of minimal area, allowing many RTI calculations to be performed in parallel.

6.1.1.2 Choice of RTI Method

There are several factors that must be considered when selecting the best fit RTI method. These factors include resources required for a pipelined implementation, any possible implementation difficulties and also any drawbacks to the method itself such as lack of precision.

To determine the resources required for a pipelined implementation we look at the number of basic arithmetic operations required to perform a single RTI test. Although many methods employ early exiting strategies when executed in software, these are not applicable when executed in a pipeline. We also include operations from forks of an algorithm, as even though execution may only flow through one path in software, resources must to be provided for all eventualities in a pipeline. The operation counts for the RTI methods investigated are presented in Table 11.

RTI Method	**Additions**	**Subtractions**	**Multiplications**	**Divisions**	**Square Roots**
Angle Summation	17	12	21	4	3
Möller-Trumbore	9	9	27	1	0
Snyder-Barr	8	9	15	1	0
Badouel	8	4	11	1	0
Plücker Coordinate	18	5	27	1	0
Edge Crossings	20	0	21	1	0
Half-Plane Testing	12	18	33	1	0
Arenbergs Method	9	6	15	1	0

Table 11: Worst case arithmetic elements required for RTI method execution.

We can see that the angle summation, half-plane testing and edge crossings methods are among the more costly in terms of arithmetic elements required. Angle summation also has precision issues. We therefore discount these methods from any further investigation.

The remaining methods, shown in Table 12, would all be suitable for pipelined implementation in hardware. To determine the best candidate, the arithmetic system used for processing must be considered.

RTI Method	Additions	Subtractions	Multiplications	Divisions	Square Roots
Möller-Trumbore	9	9	27	1	0
Snyder-Barr	8	9	15	1	0
Badouel	8	4	11	1	0
Plücker Coordinate	18	5	27	1	0
Arenbergs Method	9	6	15	1	0

Table 12: Number of arithmetic elements required in pipelined implementation for preferred RTI methods.

6.1.1.3 Choice of Arithmetic System

Like the other parameters considered, there is a choice in terms of arithmetic system used for RTI accelerator hardware implementation. The three systems considered are fixed-point, floating-point and LNS based fixed-point system, all using 32-bit words as standard.

As the RTI accelerator will be used in the synthesis of highly realistic images, a range and precision that will minimize any accumulation of error must be used, which could otherwise lead to visible artefacting. For this reason, fixed-point processing was discounted.

Floating-point and fixed-point LNS must now be considered in terms of speed and area requirement. All RTI methods investigated, including Badouels, contain an unavoidable division operation. In floating-point arithmetic, division operations are commonly implemented as iterative routines and are therefore difficult to adapt to a pipelined formation. Achieving the required speed from a floating-point implementation will be therefore be problematic and is a non-contender to start with.

However, for completeness, if we were to consider both floating-point and LNS based fixed point implementations, this raises the question of how much silicon would be required. This is of high importance, due to the push for mass parallelisation of the RTI acceleration unit to enable brute force processing.

The strength of LNS has always been with multiplications and divisions, which are essentially a simple addition or subtraction. The difficulty with LNS arithmetic has always lain in the implementation of addition and subtraction, particularly the latter, which involves the evaluation of a nonlinear function. Until recently this has resulted in implementations which are significantly slower or less accurate than their floating-point counterparts, which has nullified the advantages of multiplication and division. Recent developments however have shown that addition and subtraction can be performed with similar accuracy to floating-point implementations of the same wordlength, proven in the creation of a 32-bit fixed-point LNS based microprocessor [85].

For this comparison, we based the area of each arithmetic element required on the normalised floating-point and fixed-point LNS areas provided in the work by Haselman et al [86]. In this work, the authors perform a comparison of FPGA implementations of the basic arithmetic operations of addition, subtraction, multiplication and division for both single precision (32-bit) and double precision (64-bit) number formats. For LNS addition and subtraction, Lewis' method of interpolation between values of the non-linear functions stored in lookup tables is used [87].

The single precision implementation area results are shown in Table 13. Note that the normalised area is an indication of the number of slices required to implement a single unit.

Arithmetic System	**Addition**	**Subtraction**	**Multiplication**	**Division**
32-bit Floating-Point	424	424	368	910
32-bit LNS	1291	1491	20	20

Table 13: Normalised areas for single-precision FPGA implementations of the basic arithmetic operations in both floating-point and LNS.

Table 14 shows the normalised area requirement for floating-point and LNS implementations of the preferred RTI methods from Table 12. We can see that, for both arithmetic systems, Badouels method has the lowest total area and that, in the case of LNS implementation, the majority of overall area is composed of the addition and subtraction elements.

RTI Method	**Arithmetic System**	**Additions**	**Subtractions**	**Multiplications**	**Divisions**	**Total**
Möller-Trumbore	FLP	3816	3816	9936	910	**18478**
	LNS	11619	13419	540	20	**25598**
Snyder-Barr	FLP	3392	3816	5520	910	**13638**
	LNS	10328	13419	300	20	**24067**
Badouel	FLP	3392	1696	4048	910	**10046**
	LNS	10328	5964	220	20	**16532**
Plücker Coordinate	FLP	7632	2120	9936	910	**20598**
	LNS	23238	7455	540	20	**31253**
Arenbergs Method	FLP	3816	2544	5520	910	**12790**
	LNS	11619	8946	300	20	**20885**

Table 14: Normalised areas for implementations of preferred RTI methods.

This analysis, using the floating-point and LNS based fixed-point areas provided by Haselman et al [86], also suggests that a LNS based implementation of Badouels algorithm will have an overall area approximately 1.6 times that of a floating-point implementation. Recent work however has shown that area associated with the LNS addition and subtraction elements can be reduced by 35% in comparison to the LNS addition and subtraction elements used in the ELM [85]. These arithmetic elements have similar area to Lewis' techniques used by Haselman et al, meaning an equal or improved advantage also applies here.

In summary, we have shown that in terms of execution speed and ease of implementation, a fixed-point LNS based implementation is preferable to that of a floating-point implementation. We have also shown that, using recent developments in LNS addition and subtraction, an overall LNS area equivalent to that of a floating-point implementation is achievable. Therefore, a fixed-point LNS based RTI pipeline is our preferred option. We will base the arithmetic elements required in the pipeline on those from the work of Coleman et al [85], making any modifications required for a pipelined implementation. We will also bear in mind the recent developments in LNS addition and subtraction technology [88], as it is likely that the pipeline will be adapted to make use of such techniques to reduce overall area thus further improving scalability.

6.1.2 Use of LNS Arithmetic

As the preferred design for RTI acceleration hardware makes use of logarithmic arithmetic processing, in this section we give a brief summary of the background of LNS and previous work in this field. We then look at the arithmetic elements used in some promising recent work from Coleman et al, on which we will base our acceleration designs.

As an alternative to floating-point, a number of workers have suggested the use of a logarithmic number system for the representation and processing of real numbers. In a LNS, a value is represented as a fixed-point base-2 logarithm, allowing multiplication and division operations to be performed in one fixed-point operation time. Square and square-root operations require minimal hardware as they are performed simply by logically shifting the LNS word.

The implementation of LNS addition and subtraction has always been the major difficulty, particularly the latter which involves the evaluation of a nonlinear function. This has until recently nullified the advantages of multiplication and division.

In advanced graphics algorithms it is often the case that unavoidable operations such as square-roots and divisions are necessary, as we have witnessed in our analysis of several RTI methods. These are costly in floating-point and often have a detrimental impact on performance of the algorithm. As these normally costly operations can be performed quickly and efficiently using LNS, this raises the strong suggestion that this is considered for hardware implementations of advanced graphics algorithms such as the one considered herein.

6.1.2.1 Previous Work

The LNS arithmetic elements used in this work are based on those used by Coleman et al in the European Logarithmic Microprocessor (ELM) [85]. For this reason, we concentrate on this work, but note that there has been a great deal of earlier work on the subject of LNS arithmetic [87, 89-95]. These works range from the basics of LNS arithmetic, reduction of the size of the lookup tables required for LNS addition and subtraction, to an implementation of a 30-bit LNS processor.

Prior to the development of the ELM, Coleman introduced an algorithm to reduce the size of lookup tables required for LNS addition and subtraction [96]. It was noted that, as large ranges of the lookup function are zero, these need not be implemented. Interpolation across the table can also be used to reduce size, the most simple, first order Taylor series, requiring a table of derivatives. The same author continues with a comparison between a 32-bit LNS arithmetic unit that uses these techniques to an equivalent 32-bit floating point unit [97]. It is noted that such a system can only offer an advantage overall if LNS addition and subtraction can be performed with the speed and accuracy of a floating-point unit, even though LNS multiplications and divisions can be performed in minimal time and with no rounding error. This work resulted in a 32-bit LNS arithmetic unit which would perform substantially better in both speed and accuracy than floating-point. The exact speed improvement depends on the actual operations performed, but was approximately twofold for a ratio of additions to multiplies of 40-60%. The improvement in accuracy depends on the operands, but was generally also around twofold during simulations. The same author describes a method which allows logarithmic arithmetic to be performed at higher precision than normally available, with little additional hardware or execution time [98]. This technique requires that all data lies in a restricted range, and that all such values are scaled to the maximum range of the number system.

The European Logarithmic Microprocessor is described by Coleman et al in [99], and its operation is illustrated using a recursive least squares (RLS) algorithm. The ELM contains four single-cycle arithmetic logic units (ALU) based on the authors previous work, capable of performing LNS multiplication, division and square root operations. LNS addition and subtraction are treated as special cases, with two multi-cycle ALUs provided for their execution. A comparison is made between the ELM device fabricated in 0.18µm technology, with an industry standard floating-point DSP device from Texas Instruments also manufactured in this technology [85]. This work concluded that LNS arithmetic would be of benefit in applications involving significant complex computation.

As the achievable operating speed and accuracy of the arithmetic elements used in the ELM have been proven, these are a good choice for use in our RTI acceleration pipeline design. A description of these elements is provided in the following section.

6.1.2.2 LNS Arithmetic Elements

The data format used in the chosen arithmetic elements includes a sign bit, an 8-bit integer and a 23-bit fraction, both of which form a coherent two's complement fixed point value in the range ≈-128 to 128. This means that values in the range $\pm 2.9 \times 10^{-39}$ to 3.4×10^{38} can be represented, with a special value used to indicate zero. The equivalent floating point range is $\approx \pm 1.2 \times 10^{-38}$ to 3.4×10^{38}.

As the accuracy of this format has been proven in the ELM, our RTI acceleration pipeline will continue to make use of this data format.

It should be noted that in the adopted LNS arithmetic elements, the multiplier and divider operate without any rounding error, and the error in the addition and subtraction elements is reduced in comparison to equivalent 32-bit floating point hardware [100, 101]. There will therefore be no detrimental impact on the quality of any images generated form a RTI unit based on such arithmetic hardware in comparison to a floating-point based design.

6.1.2.2.1 Multiplication/Division

LNS multiplication and division, using the data format mentioned above, can be performed using fixed point additions and subtractions. These operations take only one fixed point operation time to complete.

6.1.2.2.2 Addition/Subtraction

Coleman et al provide an excellent description of the LNS addition/subtraction unit [97, 100].

The LNS addition/subtraction unit takes three clock cycles to complete processing, or four if the range shifter is used. The range shifter is used selectively during subtractions, transforming the operands into a linear region of the function describing the subtraction.

Modifications will be required for the unit to operate in the RTI pipeline where it will be provided new inputs on every clock cycle.

6.1.3 A Custom RTI Pipeline

In this section, we propose a design for a custom RTI pipeline, capable of performing one RTI calculation per clock cycle.

It is possible to implement a custom pipelined data-path based on the published version of Badouels method [34], but this is overly complex and will be wasteful of resources in hardware. There are several areas of the algorithm which can be restructured to allow a more efficient hardware implementation.

6.1.3.1 Restructuring the Algorithm

Pseudo code for the published version of Badouels method is shown in Figure 28. It can be seen that this version has two distinct paths and the path that is taken depends on the value of u_1. This fork makes a pipelined implementation of this algorithm complex and inefficient in terms of hardware.

```
int baduoel (triangle t,  ray r, intersection i)
{
      float u0, v0;

      i->point[0] = r->dir[0]*i->t + r->orig[0];
      i->point[1] = r->dir[1]*i->t + r->orig[1];
      i->point[2] = r->dir[2]*i->t + r->orig[2];

      u0 = i->point[t->i1] - t->vertex0_i1;
      v0 = i->point[t->i2] - t->vertex0_i2;

      if (t->u1 == 0.0) {
            i->beta = u0*t->recip_u2;
            if (i->beta < 0.0 || i->beta > 1.0) return 0;
            i->alpha = (v0 - i->beta*t->v2)*t->recip_v1;
      }
      else {
            i->beta = ((v0 * t->u1) - (u0 * t->v1))
                  * t->recip_det;
            if (i->beta < 0.0 || i->beta > 1.0) return 0;
            i->alpha = (u0 - i->beta*t->u2)*t->recip_u1;
      }
      if ((i->alpha < 0.0) || (i->alpha + i->beta> 1.0)) return 0;

      return 1;
}
```

Figure 28: Pseudo code for published version of Badouels algorithm.

In hardware, it is unnecessary to provide hardware resources for both branches as the more commonly taken branch, $u_1 \neq 0$, has adequate hardware for the two different execution paths. The inputs used however will differ, depending on the value of u_1. A reworked version of the algorithm, with the two distinct paths merged, is shown in Figure 29.

```
int badouel (triangle t,  ray r, intersection i)
{
      float uv, vu;

      i->point[0] = r->dir[0]*i->t + r->orig[0];
      i->point[1] = r->dir[1]*i->t + r->orig[1];
      i->point[2] = r->dir[2]*i->t + r->orig[2];

      uv = i->point[t->i1] - t->vertex0_a;
      vu = i->point[t->i2] - t->vertex0_b;

      i->beta = ((vu * t->A) - (uv * t->B))
            * t->C;
      if (i->beta < 0.0 || i->beta > 1.0) return 0;
      i->alpha = (uv - i->beta*t->D)*t->E;
      if ((i->alpha < 0.0) || (i->alpha + i->beta> 1.0)) return 0;

      return 1;
}
```

Figure 29: Restructured version of Badouels algorithm.

Of course, the ray-plane test must also be included, as this is how we calculate the ray-plane intersection distance, *t*. In an attempt to reduce render-time workload, the triangle data values A, B, C, D and E can be pre-calculated. These values are dependent on the value of u_1, which is also calculated prior to render time. These values are calculated as shown below.

CASE: u1 = 0	
t->vertex0_a	Vertex0(i2)
t->vertex0_b	Vertex0(i1)
t->A	1
t->B	0
t->C	1/u2
t->D	v2
t->E	1/v1
CASE: u1 ≠ 0	
t->vertex0_a	Vertex0(i1)
t->vertex0_b	Vertex0(i2)
t->A	u1
t->B	v1
t->C	1/(v2*u1-u2*v1)
t->D	u2
t->E	1/u1

Table 15: Values of Badouels algorithm variables depending on value of u_1.

Two further optimisations are possible. In the calculation of *beta*, it is obvious that the value *t->C* can be removed and incorporated into the values of *t->A* and *t->B* during pre-processing, removing the need for *t->C* and the multiply operation to which it is an input. The resulting values of *t->A* and *t->B* are shown Table 16.

The other optimisation involves *t->E* in the calculation of *alpha*. The value of *t->D* can incorporate the value of *t->E*, but then the value of *uv* also needs to be multiplied by *t->E*. This can be done concurrently with the calculation of the other components of *alpha*, which could possibly reduce the latency of the pipeline.

New Value	Calculation
t->A	t->A * t->C
t->B	t->B * t->C
t->D	t->D * t->E

Table 16: Modification of Badouels algorithm variables.

The resulting algorithm is shown in Figure 30.

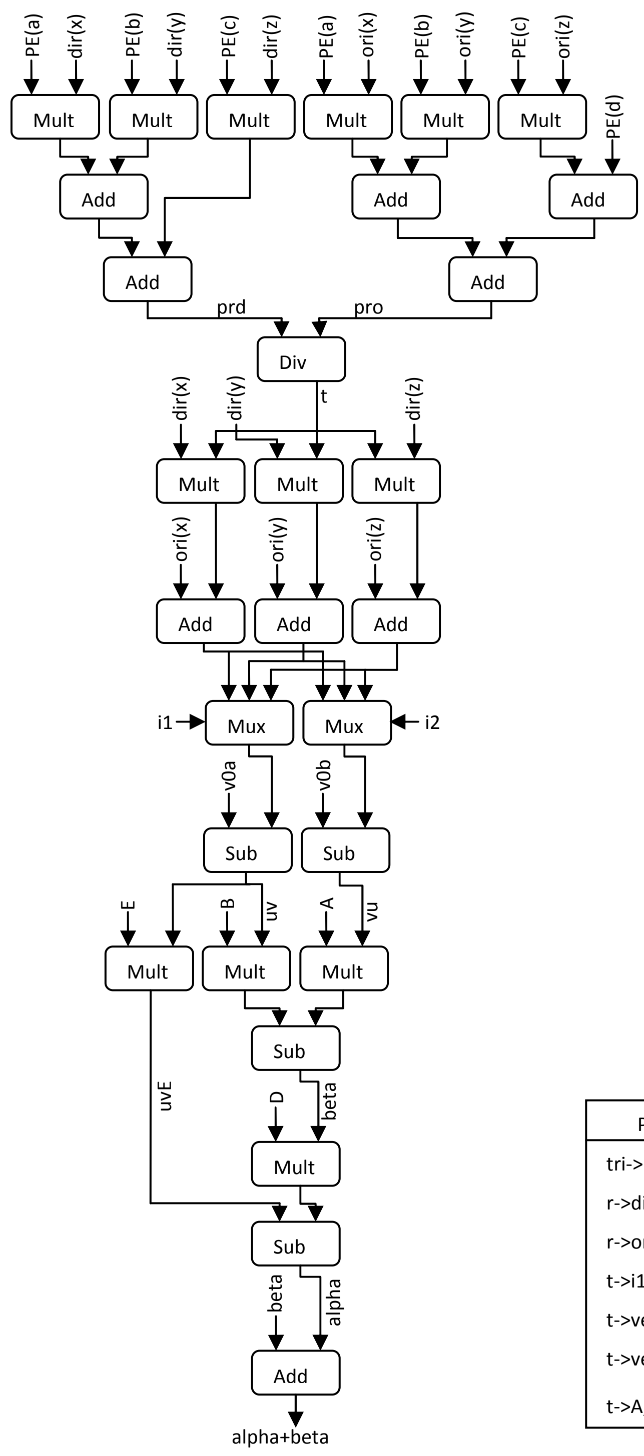
```
int badouel_inc_pay_plane (triangle tri,  ray r, intersection i)
{
      float prd, pro, t, uv, vu, uvE;

      prd = tri->plane_eq_a * r->dir[0] + tri->plane_eq_b * r->dir[1]
            + tri->plane_eq_c * r->dir[2];

      pro = tri->plane_eq_a * r->orig[0] + tri->plane_eq_b
            * r->orig[1] + tri->plane_eq_c * r->orig[2]
            + tri->plane_eq_d;

      if (prd == 0.0)
            i->t = maximum;
      else
            i->t = -pro/prd;

      i->point[0] = r->dir[0]*i->t + r->orig[0];
      i->point[1] = r->dir[1]*i->t + r->orig[1];
      i->point[2] = r->dir[2]*i->t + r->orig[2];

      uv = i->point[t->i1] - t->vertex0_a;
      vu = i->point[t->i2] - t->vertex0_b;

      i->beta = (vu * t->A) - (uv * t->B);
      if (i->beta < 0.0 || i->beta > 1.0) return 0;
      uvE = uv * t->E;
      i->alpha = (uvE - i->beta*t->D);
      if ((i->alpha < 0.0) || (i->alpha + i->beta> 1.0)) return 0;

      return 1;
}
```

Figure 30: Fully restructured version of Badouels algorithm, including ray-plane intersection calculation.

For further details on the operation of the algorithm and the variables used see section 4.1.6.

6.1.3.2 RTI Pipeline Structure

In this section we present a design for a custom RTI datapath, capable of performing one RTI calculation per clock cycle. The structure of this pipeline, based on the pseudo code presented in Figure 30 can be seen in Figure 31 below.



Pseudo Code	Diagram
tri->plane_eq	PE
r->dir	dir
r->orig	ori
t->i1, t->i2	i1, i2
t->vertex0_a	v0a
t->vertex0_b	v0b
t->A, B, D, E	A, B, D, E

Figure 31: Structure of RTI datapath.

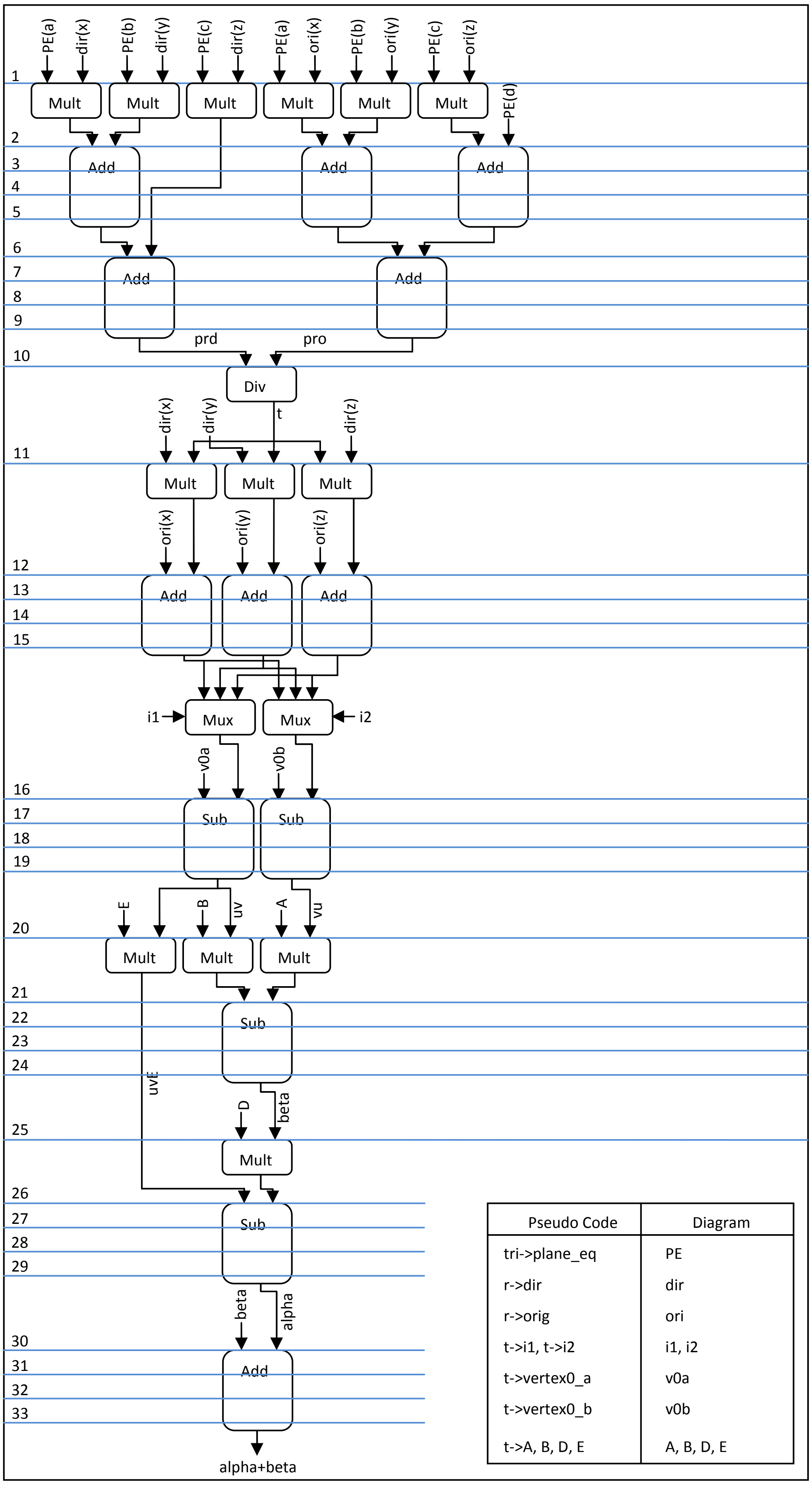
6.1.3.2.1 Integrating LNS Arithmetic

One complication lies with the addition and subtraction arithmetic units which have been designed previously for use in the ELM [100, 101], as these take multiple clock cycles to process. In their use in the ELM, these units are allowed to finish processing before new inputs are provided; these units must therefore to be altered to work in our adopted structure where they will be supplied with inputs on every clock cycle.

As described in section 6.1.2.2, the units take either three or four clock cycles to complete processing depending on whether the range shifter is used. For use in the RTI pipeline, the LNS addition/subtraction unit was modified by inserting registers between the various components of the unit ensuring that each stage would have sufficient time to complete at a typical operating speed. This resulted in a pipelined LNS addition/subtraction unit that consistently takes four clock cycles to complete processing and can accept new input on every clock cycle, making it suitable for use in the RTI pipeline.

Another feature of the LNS addition/subtraction is the use of read only memories (ROM) for the required lookup tables. As with every other element in the RTI pipeline, lookups from these ROMs will be required in every clock cycle, meaning that there is little opportunity to share ROMs between addition/subtraction units. We must therefore ensure that ROMs of optimal size and access time are used during implementation.

After adapting the structure to use the LNS arithmetic processing elements, the resulting structure and data flow of the complete RTI datapath can be seen in Figure 32. The different pipeline stages are depicted with horizontal lines, note how the addition and subtraction units span multiple pipeline stages. The horizontal lines depict the start of a pipeline stage, where values are registered for use throughout that stage.



Pseudo Code	Diagram
tri->plane_eq	PE
r->dir	dir
r->orig	ori
t->i1, t->i2	i1, i2
t->vertex0_a	v0a
t->vertex0_b	v0b
t->A, B, D, E	A, B, D, E

Figure 32: Structure of RTI datapath depicting pipeline stages.

6.1.3.3 RTI Pipeline Operation

In operation, the pipeline will be supplied with ray and triangle data. If a RAS is used, the ray will be tested against a subset of scene triangles within the pipeline, alternatively with no RAS technique employed all scene triangles will be tested for intersection with each ray.

The function of the RTI pipeline is to find the first (i.e. closest to the ray origin) triangle successfully intersected by a particular ray. To store the first triangle intersected, we must check to see whether a closer triangle has previously been intersected, and also whether the current test is valid.

6.1.3.3.1 Validity Testing

Instead of gating the clock to halt the processing of the RTI pipeline when there is no valid input, our simpler approach is to add a validity flag, which would inhibit the overwriting of any result data when set.

Figure 31 and Figure 32 do not show the validity testing that is present throughout the pipeline. To hold the result of the RTI result (hit/miss), and also test validity, simple bit flags representing these values can be cascaded through the pipeline alongside the other data values and updated when necessary. On the completion of a RTI calculation, the combination of validity and RTI result flags will tell us whether we have found a valid triangle intersection or not.

6.1.3.3.2 Finding the First Triangle Intersected

As mentioned, the result status (hit/miss) of the RTI is updated as it is cascaded through the RTI pipeline. As the processing occurs in a pipelined fashion, and there is no opportunity for early exiting, the result status flag is set to *true* at the beginning of processing. If, at the end of processing a particular triangle against a ray, the value is still *true*, the ray has intersected the triangle. If, at any point throughout the pipeline, the result status is *false* then the ray does not hit the triangle and this value is not updated by subsequent tests.

There are several stages where a ray can be proven not to hit a triangle, such as after the calculation of the ray-plane intersection distance, *t*, and also at the calculation of the barycentric coordinates, *alpha* and *beta*, and the sum of these two values.

For a valid intersection, *alpha*, *beta* and *alpha+beta* must lie in the range $0 \leq x \leq 1$. Testing these values is very simple in the LNS representation used. To test if a LNS value is less than zero, we can look at the most significant bit which contains the sign of the value. If this bit is set, then the value is less than zero. To test if the value is less than unity is just as simple, as the bottom 31 bits of the LNS word are a fixed point signed representation of the value itself. As the value unity is represented as zero in the LNS fixed point format employed, we can again test a bit to see if the value is less than unity, this time bit 30. If this bit is found to be set, then the value is less than unity.

The only data required after a ray has been processed are details on the intersection with the triangle closest to the ray origin. The information about this intersection must therefore be stored for later use. In the design, this data is stored in registers at the end of the pipeline and overwritten when appropriate. To check if the current value of the ray-plane intersection distance, *t*, is less than the currently stored value, a simple signed greater than test taking the 31 least significant bits of both values as inputs can be used. The same test can be used to test *t* with a small positive *epsilon* value. This ensures that the intersection distance is not negative, and also that surface that a ray is fired from (for reflected or transmitted rays) is not intersected again, which can be a common problem in software due to rounding errors.

As we compare *t* with a previous value, we must always reset the stored value to a large 'maximum' value when we commence processing a new ray. We will use the maximum positive value that can be represented in the LNS number format employed.

6.1.3.4 Memory Options

As one RTI test will be performed per clock cycle in the pipeline design, triangle data must be stored in a manner where it can be accessed quickly. Due to the bandwidth required to transfer data for one triangle per clock cycle, this data must be stored on-chip in RAM.

There is a choice in which variety of RAM to use. Dynamic RAM (DRAM) performs well when reading many contiguous addresses, but not so well when reading from randomly distributed addresses. This would make DRAM the ideal choice if no RAS is employed, as all triangles stored must be tested, and these can be read from consecutive memory addresses. As the RTI acceleration unit can also operate using a RAS, we must also consider how DRAM would perform in such a scenario.

It is common, and our adopted practice, to store the data for all triangles contained within one particular node, voxel or volume of a RAS consecutively in memory. The pointers to the triangle data from the containing volume are then only the starting address of its contained triangles, and the number it contains. This means that the number of triangles tested which are stored in consecutive memory locations is dependent on the number of triangles stored per node, voxel or volume if a RAS is employed, which can vary greatly. We therefore conclude that DRAM is not the ideal choice in this application, as even though its average access time is good when reading long runs of consecutive addresses, the unit may only need to read two or three triangles from memory and then jump to a new address. The variability in access time could also make integration with the pipeline, with fixed clock cycle time, problematic.

The other choice of memory considered is static RAM (SRAM), which has consistent access times whether reading from neighbouring or disparate addresses. Due to the variability in the number of read operations from consecutive addresses, this seems to be a good fit.

It is not only the type of memory which is an important factor, but also its width and how it is integrated into the pipeline. Of course, as we are processing one RTI per clock cycle, all values contained for a particular triangle must be read during each clock cycle, but there are several options on how to integrate this into the design.

One simple option would be to read all values at the start of the pipeline, meaning we have all values on hand for when they are required. There is, however a drawback to this approach as we will have to buffer values until they are used which could be costly in terms of hardware registers, especially for values which are only required towards the end of the pipeline.

Another approach is to either get several batches of triangle data at different stages through the pipeline, or receive each value as it is required. Both of these techniques reduce the number of registers required and the overall size of the pipeline. A drawback of the latter approach would be the more intricate storage of the triangle data in memory initially, which is a trivial problem to solve during the pre-processing of the triangle data.

The method used in the RTI pipeline is that of receiving batches of triangle data at multiple pipeline stages due to its simplicity.

6.1.3.5 RTI Pipeline Latency

As displayed in Figure 32, the RTI datapath has many pipeline stages. This means that the execution of the many partially completed RTIs for a certain ray will need to be finished when no other triangles remain to be tested against a particular ray. As we do not gate the clock input to the pipeline, these partially completed RTIs will continue to be processed with their results stored as necessary.

When there are no further triangles to test for a ray, the validity flag produced by the input control structure will not be set. This means that whatever inputs are provided to the RTI pipeline, the currently stored result will not be overwritten as any invalid data is cascaded through the pipeline.

Although we have solved the problem of data integrity using a test validity flag, the issue of efficiency must now be raised. If processing many thousands of triangles against a ray, likely to be the case if no RAS technique is employed, waiting an extra 34 clock cycles for the pipeline to flush will add a negligible amount to the overall ray processing time. If however we are only processing two or three triangles per ray, which may be the case if a RAS is used, then waiting for the pipeline to flush for the final results could increase the ray processing time by tenfold.

One approach to alleviate this problem is to have multiple result registers at the end of the RTI pipeline, and a ray identifier, which is also cascaded through the pipeline along with the validity and result status flags. If a triangle is found to be intersected by the current ray, the intersection results are stored in the result registers corresponding to this ray using the current ray ID as an index.

This solution would allow the pipeline to continue processing a new ray, while waiting for any partially processed RTIs to be flushed. Although we would still have wait for any ray to finish being processed before we could access the results, the pipeline could be utilised during this time by processing RTIs for another ray. This results in improved unit efficiency, especially when dealing with small numbers of triangles per ray which could be the case if a RAS technique is employed.

6.1.3.6 RTI Pipeline Scalability

There are two different options available for scaling the RTI pipeline design:

1. Use multiple RTI pipelines to process the same ray.
2. Use multiple parallel RTI units to process multiple different rays.

In the case of the first option, all RTI pipelines would require access to only a subset of the scene triangles, and would process many RTIs for the same ray in parallel, thus reducing processing time for a single ray. If no RAS is used, each pipeline would hold an equal proportion of the total scene triangles. If a RAS is needed, pipelines could store triangles contained in pre-designated volumes or nodes (ideally non-overlapping to avoid RTI test repetition between units, however overlapping volumes would not cause problems).

As the processing for a single ray will be split across multiple pipelines, the latency of the pipelines could add to the processing time of the reduced number of RTIs processed per unit. The solution presented in the previous section would be useful in this situation, improving utilisation and therefore efficiency of the pipelines. A negligible amount of processing would be required after completion of the RTI tests to determine the smallest ray-triangle intersection distance returned from all pipelines. This approach fits perfectly with our vision for a scalar unit, capable of processing all RTI tests for a single ray as quickly as possible.

The fact that in photon mapping there is often a serially dependency from one ray to the next complicates an implementation of the second option, where there must be a pool of rays ready to be processed. An alternative use of this second option, however, would be to use multiple renderers each processing only a portion of the final image and using one of the multiple parallel RTI pipelines.

In both cases, the number of RTI tests performed per clock cycle will increase, and in the case of the first option, the overall time to process a single ray will decrease. Diagrams of both parallelisation options are shown in Figure 33 and Figure 34.

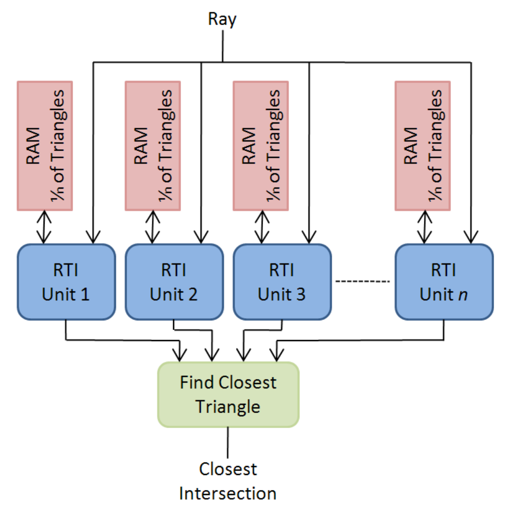


Figure 33: Diagram showing parallelisation option for processing a single ray.

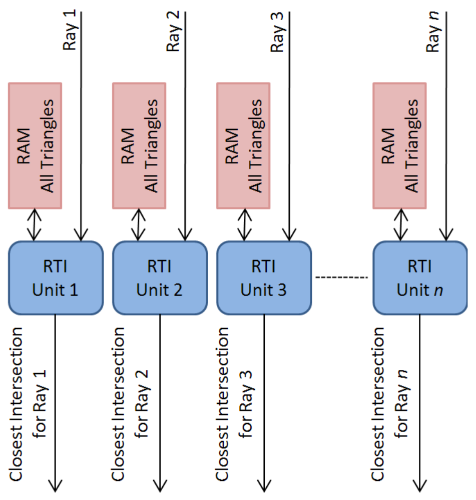


Figure 34: Diagram showing parallelisation option for processing multiple different rays.

We see more promise with the first of these approaches. In this case, as well as the faster per-ray processing which is useful when dealing with serially dependant rays, the triangles making up the scene can be distributed throughout the multiple RTI units, reducing the memory requirements of each. This is not the case where the units process numerous different rays, where each unit must contain all triangles. Such an approach may affect the scalability of the solution due to increased hardware size caused by the rise in memory requirement.

6.1.3.7 RTI Pipeline Integration

The RTI pipeline, or more likely multiple parallel pipelines, simply perform RTI calculations with the data they are provided. Some additional control is required to be able to process batches of triangles for a particular ray.

As it is common for many triangles to be tested against a single ray, a control structure which holds the ray under test constant is needed while all necessary triangles are tested against this ray. This control structure would vary the address in memory of the triangle to be tested on every clock cycle, meaning that correct triangle data is available throughout the pipeline for the RTI tests. The control unit would continue incrementing the triangle address until all triangles have been tested. Each pipeline used would require its own control unit.

Inputs to such a control structure would be the ray data, along with the local starting address of the triangles under test. For a stopping condition, the number of triangles to be tested must also be provided. This makes the pipelines suitable for use with or without a RAS technique. The ray data, the memory address of the current triangle being tested, and also the validity of the test would be outputs of this control block and would feed the RTI pipeline. The validity flag, as described earlier, would be used to enable the pipeline to flush without overwriting the RTI results registered at the end of the pipeline.

As a higher level, the RTI pipeline(s) (and control structure(s)) could either interface to a PC running the main-line renderer through a high speed link (USB 3.0 or IEEE 1394), or to a general purpose microprocessor running the main-line renderer which would allow for improved interfacing.

6.1.4 Summary of RTI Acceleration Hardware Design

After investigating various options in terms of RTI method, hardware structure and arithmetic system, we presented a design for the acceleration of RTI calculations. The design is based on a modified Badouels algorithm, making use of LNS arithmetic in a pipeline structure and is capable of performing one RTI calculation per pipeline clock cycle.

The pipeline is based on LNS arithmetic elements used in the European Logarithmic Microprocessor [85, 100, 101]. These elements have been proven to operate successfully at 125MHz when fabricated in 0.18µm technology. We are confident that if the pipeline is implemented using a smaller technology, then higher operating clock rates are achievable.

We have also shown how the number of pipelines can be scaled. The option most applicable to photon mapping acceleration is the use of multiple pipelines used to process the same ray, which can offer reduced memory requirement per pipeline as well as reduced processing time per ray. The number of RTI calculations that can be performed per clock cycle scales linearly with the number of pipelines used; meaning that for relatively few pipelines significant speedups could be achieved. Notably, due to this scalability, it is likely that any advantage gained can be maintained in the presence of ever increasing processor speeds.

6.2 Photon Search Acceleration Hardware Design

In this section we present a design capable of accelerating the photon search function, or spatial point range searching for other applications. This hardware has a massively parallel arrangement, capable of determining whether multiple photons are within a search volume simultaneously [102].

Like the RTI acceleration hardware design, this design is also scalable to the point that several devices, each containing multiple parallel processing elements can be chained together.

We begin this section by looking at the various design decisions made, such as the choice of hardware structure and search volume, before looking in depth at the operation of the PS acceleration unit.

6.2.1 Design Options

There are two main choices that must be made in the design of acceleration hardware for the PS function. These are:

- Hardware structure
- Search volume

Like the RTI design, execution speed is the main driver behind our design decisions. For any hardware produced to fulfil its purpose, it must offer significant acceleration over our software implementation of the photon search.

6.2.1.1 Choice of Hardware Structure

A balanced *k*D-tree is a commonly used structure for a photon map in software. One option for acceleration hardware would be to create tree traversal units and persist with the *k*D-tree structure for the photon map. While it would be possible to provide some acceleration with this setup, the size of the hardware required would limit the scalability of the solution.

A pipeline structure, like that used in the RTI acceleration hardware design, is another option. In such a pipeline we could implement a test to check whether a photon was inside any particular search volume. This would mean that it is possible to test whether a single photon resides within this volume per clock cycle, however every photon stored would have to be tested, unlike the *k*D-tree tree traversal option where spatial regions can be eliminated from the search. To achieve any acceleration over a software implementation using a pipelined test, many pipelines would be required. As these would likely need a fair amount of arithmetic hardware, scalability would again be limited.

Another option would be to again take a brute-force approach to the photon search problem, where every stored photon must be tested, however do this by reusing hardware elements unlike a pipeline. With this approach scalability would be increased due to the reduced hardware requirement, however to achieve a massively parallelisable architecture a suitable search volume and arithmetic hardware suitable to perform the required photon inclusion testing must be found. The arithmetic hardware must be of minimal size to facilitate the scalability that would be needed to achieve sufficient acceleration for the increased number of photon inclusion tests that would have to be performed when compared to the traditional smart software approach, as shown in Table 17. This structure is our preferred option.

Number of Photon Inclusion Tests	**Model 1**	**Model 2**	**Model 3**	**Model 4**
Smart Approach (*k*D-tree RAS)	$8.950x10^{8}$	$1.146x10^{9}$	$1.160x10^{9}$	$1.222x10^{9}$
Brute Force Approach	$1.509x10^{11}$	$1.796x10^{11}$	$1.905x10^{11}$	$2.021x10^{11}$
Times Increase	***168.560***	***156.673***	***164.313***	***165.330***

Table 17: Number of photon-search volume inclusion tests performed in Smart and Brute Force approaches.

6.2.1.2 Choice of Search Volume

The scalability of a brute-force based PS acceleration unit is critically dependant on the size of the arithmetic elements required for the processing of photon inclusion tests. Therefore, several point-volume inclusion tests were investigated to determine the processing requirement of each.

The volume inclusion tests looked at were point-in-sphere and point-in-AABB. To test whether a photon is within a sphere, the distance from the photon to the origin of the sphere is calculated and compared to the sphere radius. Pseudo code for this test is shown in Figure 35.

```
int Photon_In_Sphere (Photon p, Sphere s)
{
      float squared_length;
      squared_length =  (p->x - s->x)*(p->x - s->x) +
            (p->y - s->y)*(p->y - s->y) +
            (p->z - s->z)*(p->z - s->z);

      if (squared_length <= s->squared_radius) return 1 else return 0;
}
```

Figure 35: Pseudo code for a sphere inclusion test.

To perform a point-in-sphere test in hardware, the minimum arithmetic hardware requirement would include an adder/subtractor and a multiplier which would be reused. Having this amount of hardware to perform a single photon search prohibits large scale replication.

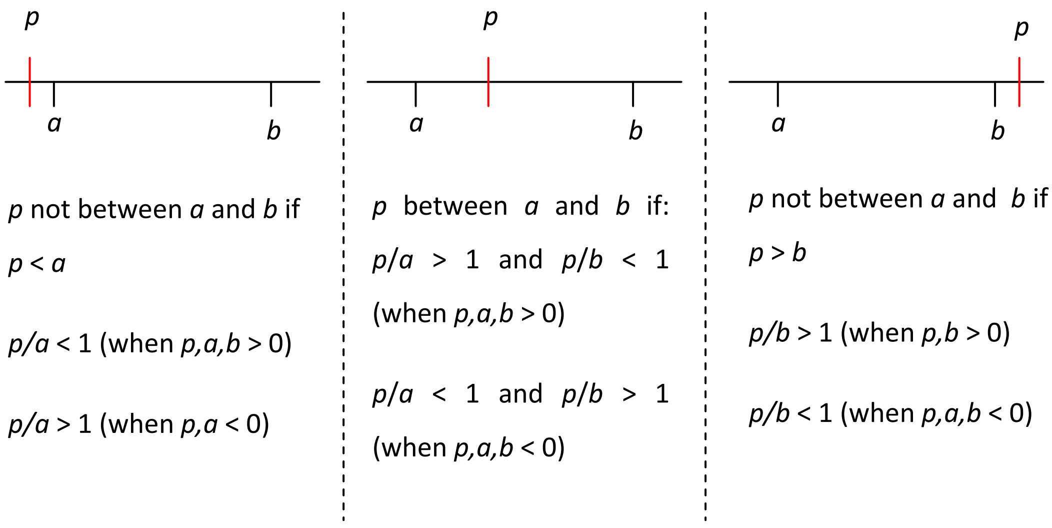
An alternative volume test that would require less arithmetic hardware to perform the required photon inclusion processing was sought. The use of an AABB as a search volume was investigated. To test for photon inclusion within an AABB, the distance from the photon to the centre of the box can be calculated in three dimensions, and then these values can be compared to the deflections, from the origin, of the box boundaries. Calculating the distance from the photon to the centre of the box, and comparing these values to the box deflections, can both be performed using a single subtractor.

There is an issue with this approach however, in that the size of a floating-point subtractor (or the alternative of a LNS based subtractor, as used in the RTI pipeline) is fairly large, compared to say a fixed-point adder or subtractor. Although an improvement on the sphere inclusion test, the size of the hardware required would still limit replication.

To get around this problem, a box can be defined by its upper and lower bounds in each dimension. This would enable photons to be tested for inclusion by comparing their position, in each dimension, to the upper and lower bounds of the box. To find these bounds, subtractions are still needed, but this calculation can performed external to the PS acceleration unit. This improved box definition would contain the upper and lower boundaries in 3-dimensions, and would only mean a small increase in the data required to describe an AABB, from four words to six.

To perform the photon in AABB test, sign checks and magnitude comparisons can be used. Sign checks are simply undertaken. It is common to execute a magnitude comparison using the subtract operation, comparing the result with zero. It is also possible to perform magnitude comparisons using divide operations, comparing the result with unity. This is not a widely used technique due to the expensive nature of divisions in floating-point architectures, but divisions can be performed quickly in LNS. In both arithmetic systems the only hardware required would be a fixed-point subtractor, which forms the basis of a floating-point comparator and also a LNS divider. This makes a massively parallel photon search using an AABB as the search volume feasible.

As LNS arithmetic has already been used in the design of RTI acceleration hardware, we continue to use this arithmetic system in the PS acceleration unit design. Having both systems using the same representation of values may be beneficial in their integration. Three cases of a photon AABB inclusion test for a single axis, using LNS division for the required magnitude comparisons, are shown if Figure 36.



p not between *a* and *b* if

$p < a$

$p/a < 1$ (when $p,a,b > 0$)

$p/a > 1$ (when $p,a < 0$)

p between *a* and *b* if:

$p/a > 1$ and $p/b < 1$ (when $p,a,b > 0$)

$p/a < 1$ and $p/b > 1$ (when $p,a,b < 0$)

p not between *a* and *b* if

$p > b$

$p/b > 1$ (when $p,b > 0$)

$p/b < 1$ (when $p,a,b < 0$)

Figure 36: AABB point inclusion test using divisions for magnitude comparison.

Pseudo code is presented in Figure 37 for the AABB inclusion test. LNS divisions are used for the required magnitude comparisons. As we can see, the use of LNS arithmetic has additional benefits in the many value comparisons with zero and unity, as these are simple check of bits 31 and 30 in the number representation used.

```
int Photon_In_AABB (Photon p, AABB c)
{
      //perform sign checking first
      if ((c->max_x < 0 && p->x >= 0)||(c->min_x >= 0 && p->x < 0))
            return 0;
      if ((c->max_y < 0 && p->y >= 0)||(c->min_y >= 0 && p->y < 0))
            return 0;
      if ((c->max_z < 0 && p->z >= 0)||(c->min_z >= 0 && p->z < 0))
            return 0;
// X axis
      if (p->x >= 0 && c->max_x >= 0)
      {
            if (p->x/c->max_x > 1) return 0;
      }

      if (p->x < 0 && c->max_x < 0)
      {
            if (p->x/c->max_x < 1) return 0;
      }

      if (p->x >= 0 && c->min_x >= 0)
      {
            if (p->x/c->min_x < 1) return 0;
      }

      if (p->x < 0 && c->min_x < 0)
      {
            if (p->x/c->min_x > 1) return 0;
      }
// Y axis
      if (p->y >= 0 && c->max_y >= 0)
      {
            if (p->y/c->max_y > 1) return 0;
      }

      if (p->y < 0 && c->max_y < 0)
      {
            if (p->y/c->max_y < 1) return 0;
      }

      if (p->y >= 0 && c->min_y >= 0)
      {
            if (p->y/c->min_y < 1) return 0;
      }

      if (p->y < 0 && c->min_y < 0)
      {
            if (p->y/c->min_y > 1) return 0;
      }
// Z axis
      if (p->z >= 0 && c->max_z >= 0)
      {
            if (p->z/c->max_z > 1) return 0;
      }

      if (p->z < 0 && c->max_z < 0)
```

```
       {
              if (p->z/c->max_z < 1) return 0;
       }

       if (p->z >= 0 && c->min_z >= 0)
       {
              if (p->z/c->min_z < 1) return 0;
       }

       if (p->z < 0 && c->min_z < 0)
       {
              if (p->z/c->min_z > 1) return 0;
       }

       return 1;
}
```
Figure 37: AABB inclusion test pseudo code using divisions for the magnitude comparisons.

6.2.2 A Custom PS Acceleration Unit

In this section we propose a design for a hardware unit to accelerate the processing of the photon search function of the photon mapping algorithm. This design takes a brute-force approach to the problem, testing all stored photons to determine if they are located within a specified AABB using a minimal amount of processing hardware, even as little as a single LNS divider. This processing is performed in a massively parallel arrangement.

There are often many thousands of photons stored during a rendering, meaning that it is not feasible to have a processing element based on an LNS divider for each of these photons. It is possible, however, to have multiple processing elements and have each sequentially process several photons. All of the location data for these photons must be stored locally on the PS acceleration unit due to the required bandwidth, with RAM being the obvious choice for this storage. In this case, rather than using RAM with multiple external processing elements, a particularly attractive option is content-addressable memory (CAM). In a CAM, values are addressed by a particular key associated with each item stored. If we are able to store the locations of photons, then the key to address these photons could be the volume we are searching.

6.2.2.1 PS CAM Structure

CAM is the hardware equivalent of an associative array in software and is particularly suited to high speed searching applications. A CAM is made up of many cells, each containing some data value(s). Every value stored in a CAM cell is associated with a key. When this key is provided to the cells, a list of the cells holding the associated value is returned, sometimes along with the value itself.

As each cell in a CAM has to decide whether the value it contains is associated with the key, some amount of local processing is needed. In all situations, the size of the processing hardware must be minimal due to the large number of cells required. This fits well with our use of an AABB as the search volume, as only a fixed-point subtractor based LNS divider would be required per cell for processing.

In a photon search CAM implementation, the data stored in each cell would comprise of only the location of a stored photon in three dimensions (all other photon data would be held in host renderer memory). The key presented to the cell would be the AABB search volume. The local processing in each cell would then, using the stored photon position and the search volume, calculate whether the photon resides within this volume.

We have already shown that it is possible to perform a photon inclusion test using an AABB as the search volume using a single divider to perform the necessary magnitude comparisons. For an increase in processing speed, while having a detrimental effect on silicon area requirement, it is possible to use multiple dividers to perform several magnitude comparisons simultaneously. As each LNS division takes one clock cycle, testing a photon for inclusion in an AABB can be completed in a maximum of six clock cycles if a single divider is used, reducing to a single clock cycle if six dividers are provided.

Each CAM cell contains at least one divider as described above, plus some control logic. The control logic within each cell deals with input and output to and from the cell, and also handles memory addressing. This memory addressing can be as simple as supplying consecutive addresses to the RAM embedded in the CAM cell if six subtractors are used, but as it is possible to use fewer than six subtractors early exit strategies can be employed.

When an early exit is possible, detected using the output of the magnitude comparison, a decision is made on whether the next data for the same photon or data for the next photon to test should be fetched. As the majority of stored photons will not be within the search volume during any particular search, the use of early exiting is beneficial.

The basic structure of a photon search CAM cell is shown in Figure 38. The CAM itself is composed of many of these cells, as shown in Figure 39.

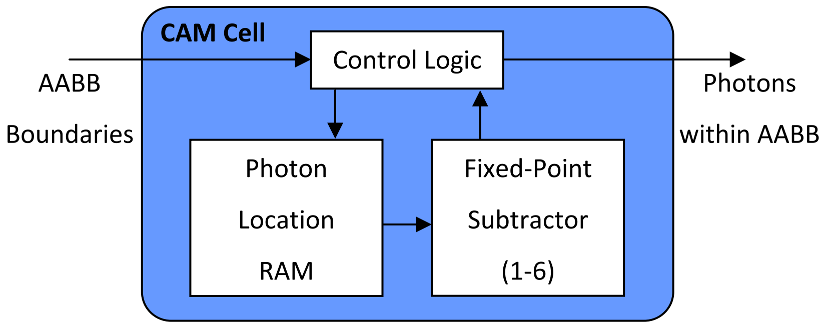


Figure 38: Basic structure of photon search CAM cell.

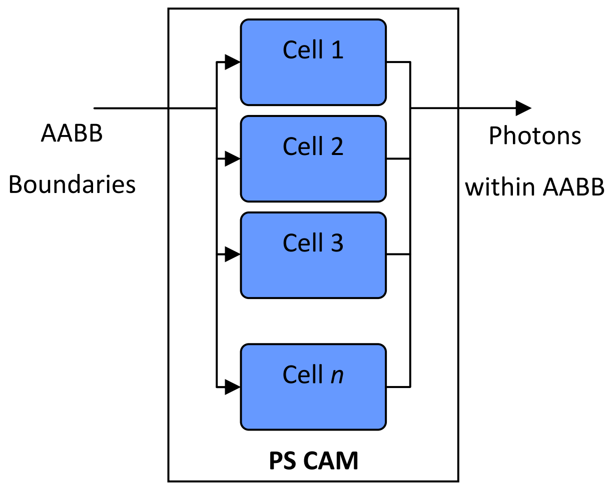


Figure 39: Structure of photon search CAM showing multiple processing cells.

6.2.2.2 PS CAM Operation

Prior to the commencement of a search, the photons locations must be stored in the PS CAM. The only data stored for any particular photon is its location in three axes, requiring 96-bits of data. A photon identifier is not stored, minimising the RAM required per cell. Instead, each cell has a search result bit vector, with a single bit representing each photon contained – the actual bit depending on the order that the photons are written to the CAM.

The CAM search operation begins with the AABB boundaries, in three dimensions, being broadcast to all cells simultaneously. Then, using these boundaries, the photon locations stored within each cell are processed sequentially. The identity of any included photons found within a cell is recorded by setting the corresponding bits in the search result bit vector.

The actual processing performed in a CAM cell is dependant on the number of LNS dividers contained. If one divider is used, there will be a maximum of six divisions to process sequentially. There is however a benefit to this approach, as if a photon is found to be outside the search AABB during an early test, the remaining sequential processing of this photon can be ignored giving rise to early exit strategies as previously described. A flow chart showing the processing for the single LNS divider option, including the early exit opportunities can be seen in Figure 40.

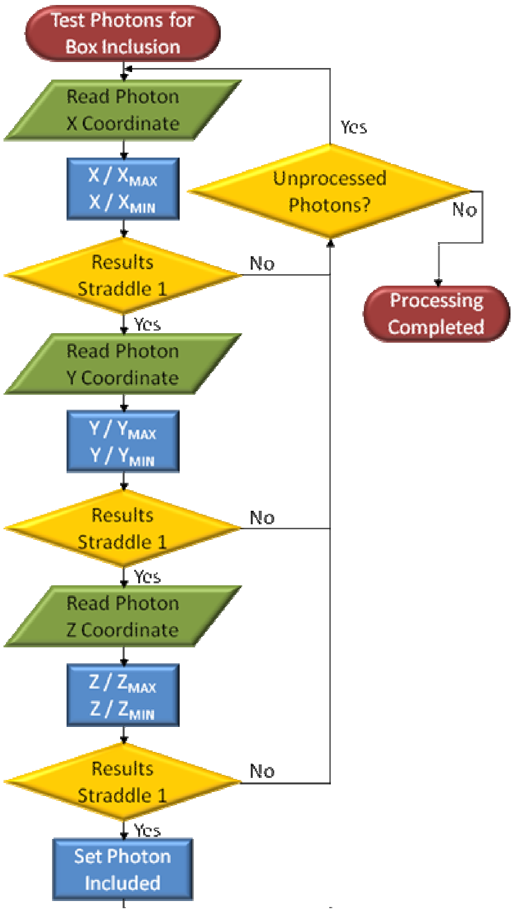


Figure 40: Flowchart detailing single divider CAM processing.

Another option would be to use two LNS dividers, and test photons for inclusion in any particular axis at once. This would double the processing hardware required, but half the processing time for included photons. As with the single divider option, early exits are possible if a photon is found to lie outside the search box in an early test.

With both of the methods discussed so far, the location of a photon is only required in one axis at any time. If the photon passes a test, then location data for the next axis can be retrieved and tested. This means that only a 32-bit memory interface is required for photon data.

Other options include using three or six dividers per CAM cell. With three dividers, the location of a photon would be tested against the lower bounds of the AABB in three dimensions concurrently, followed by the upper bounds. As with the previously mentioned tests, there is an early exit available if the photon is found to lie outside the cube after the first test. With six dividers, photon locations are tested against upper and lower bounds at the same time, resulting in the fastest execution time.

Both of these strategies will yield faster photon inclusion testing, at the price of increased processing hardware. To take advantage of this increase in processing speed, memory bandwidth would need to triple, to provide the photon locations in all three axes concurrently. A flowchart showing the processing in a CAM cell containing six LNS dividers is presented in Figure 41.

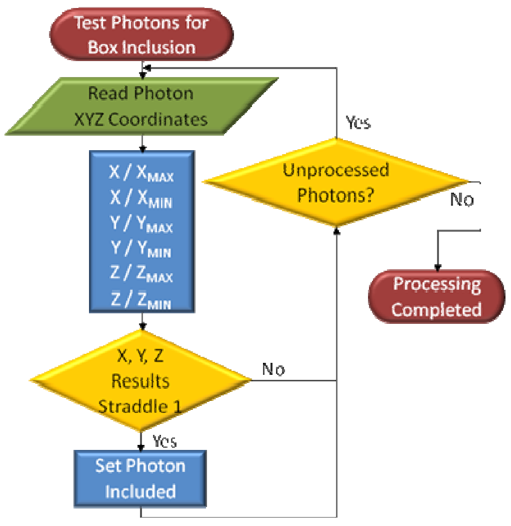


Figure 41: Flowchart showing 6-divider CAM processing.

After the search, the search result bit vectors detailing the photons found within the AABB are read back from the CAM cells in the same order as the photons were written. As the write order and the specific CAM cell being read from are known, this allows photons to be located in host renderer memory, where values such as intensity can be fetched and used in the subsequent irradiance calculation.

To reduce the data being transferred, an initial bit vector is returned to the host renderer, with each bit representing whether or not a CAM cell contains at least one valid photon. This eliminates the read-back of search result bit vectors from cells which contain no included photons.

To allow a new photon search to begin while the results from a previous search are being read, a simple double buffer arrangement can be used. This ensures that no values in a result set are overwritten mistakenly, however allows increased utilisation of the CAM.

6.2.2.3 Photon Distribution Options

If the afore mentioned technique of only reading search result vectors from CAM cells containing at least one included photon was implemented, there would be benefit in storing photons based on spatial locality. If spatially close photons are stored in the same CAM cell, there is a high probability that few cells will have many photons within the search volume, with the others having few or none. In this case the data that must be sent during result read-back is reduced.

However, in the case when fewer than six LNS dividers are used, there is a drawback to storing spatially close photons in the same CAM cell. With fewer than six dividers early exiting from testing a particular photon is possible, meaning that testing a photon within the bounds of the search AABB takes the longest time to complete. If multiple photons within the search volume are stored in one or few cells, which is likely if photons are distributed based on spatial position, this would increase the sequential processing required in these cells. Waiting for these cells to complete their processing would increase the overall processing time of the CAM.

6.2.2.4 PS CAM Issues

There are two possible issues with the choice of bounding volume and processing used for a CAM photon search. There issues relate to *k*NN searching, and the post search photon density calculation used to calculate an irradiance estimate.

6.2.2.4.1 *k*NN Searching

As the search volume is fixed during any particular search, and photon location to search origin distance is not calculated, true *k*NN searching is not possible. However, being able to perform *k*NN searches would be advantageous due to the visual improvements achievable in the rendering of caustics.

One solution to this problem would be to use a separate photon map for caustics. In such an approach caustic photons that are reflected or refracted by glossy surfaces before intersecting diffuse reflectors are stored in a caustic photon map. This approach has also been used in software [2]. During a rendering, irradiance values are formed using components from both the caustic photon map and global photon map, which stores the remainder of photons. Although the use of a separate CAM for a caustic map would not allow for dynamic resizing of the search volume during rendering, a smaller volume than that used when searching the global map can be used. With well chosen search volume sizes, caustic blurring can be minimised.

The preferred solution is similar to the commonly used software approach, where the *k*D-tree search results are refined to be within a sphere and added to a max heap to find the *k* nearest to the search origin. Instead, the distance to the search origin of the photons returned from a CAM search can be calculated by the host renderer. These distances could then be sorted using a simple sort algorithm, with the *k* photons having the smallest distances being the *k* nearest photons to the search origin. As there is no feedback into the CAM search there is no need for a max heap as used in the software approach.

6.2.2.4.2 Photon Density Calculation

Although not part of the photon search, the density calculation is also part of the overall irradiance calculation and is worthy of mention here. A precise irradiance estimate is required for realistic photon mapped images. Accuracy in the irradiance estimate ensures that the introduction of noise into the estimate is minimal, where even a small amount of noise in a sample could lead to artefacting in the final image.

When an irradiance estimate is calculated at a point on a surface, this surface is assumed to be locally flat. In the software calculation of photon density, this assumption allows the use of the area of a circle during the density calculation, as photons that are found in the *k*D-tree search are refined to be within a sphere. No matter what orientation the surface is in, the area of intersection with this sphere is circular with radius equal to that of the search radius. This is shown in Figure 42.

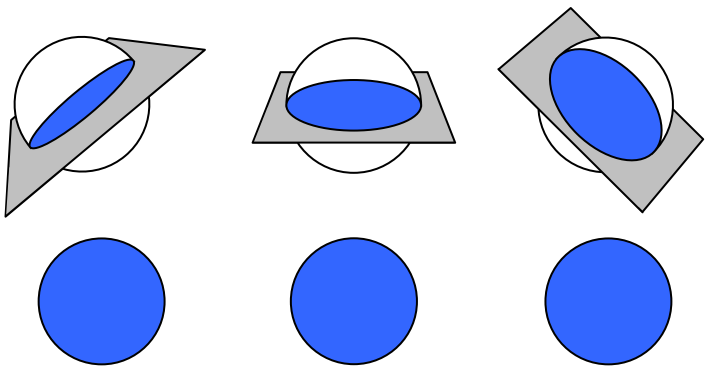


Figure 42: Cases of intersections of arbitrarily oriented planes with a sphere (top). Notice that the area of intersection in each case is circular and has same area (bottom). The plane passes through the centre of the sphere in each case.

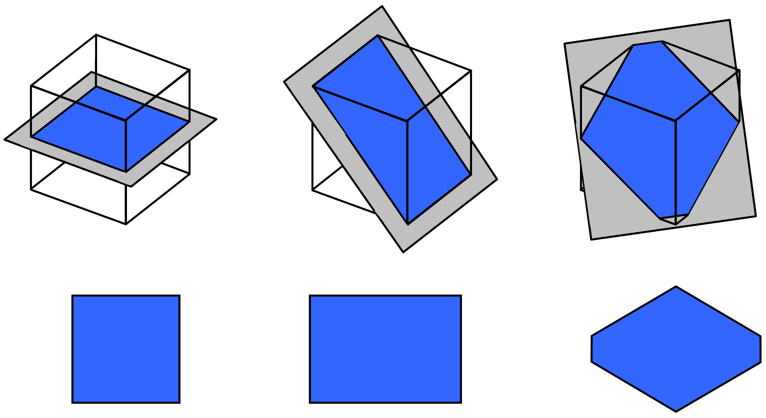


Figure 43: Cases of intersections of arbitrarily oriented planes with an axis-aligned bounding AABB (top). The plane passes through the centre of the box in each case. In the case of a cube with side length a, the area enclosed on the plane can range from a^2 to $\sqrt{2}a^2$. The area enclosed in each case forms either a 4 or 6 sided polygon on the plane (bottom).

When search volumes other than a sphere are used, the area of the assumed locally flat surface enclosed by the search volume can vary, as shown in Figure 43 for a cube. As the photon search CAM uses the AABB as its search volume, we must consider how the varying surface area enclosed by the search volume is calculated for use in the photon density calculation.

There are three plausible options that could be used to estimate the surface area enclosed by an AABB. These are:

- Use a simple fixed estimate of the surface area enclosed, somewhere between the maximum and minimum.
- Calculate the exact surface area enclosed by the cube and use this area in density estimations.
- Refine the search results in software, culling photons in the corners of the cube and leaving only those which are also enclosed in a sphere. This would allow the use of a fixed circle area in density calculations, identical to the proven software approach.

Testing has shown that the option of using a fixed estimate of enclosed surface area introduced noise into the rendered image, making this approach unsuitable for the synthesis of highly realistic images.

Calculating the exact area enclosed and using this in the density calculation also gives rise to image artefacting. This is caused by the variability in area used in irradiance calculations for spatially close surface locations, and also the fact that photons in the corners of the search AABB can have an impact on locations far from themselves. Testing shows that there is slight improvement over the use of a fixed area, however there is still noticeable artefacting in the final image making this method again unsuitable for the creation of images requiring a high degree of realism. Another disadvantage of using the actual enclosed area is the expense of the area calculation. Firstly, the four or six sided polygons representing the enclosed are must be found, then the area of this polygon must be calculated. For the large number of irradiance estimates required during a rendering, this expensive calculation could add significantly to the computation required.

The option of refining the photons returned from the CAM search so that they reside in a sphere is the preferred approach. This technique has been proven to work with the photon mapping algorithm in software. The use of a sphere allows a fixed circle area to be used during photon density calculations. This refinement is already performed in the main-line renderer after the *k*D-tree photon search, so no additional computation will be required. In fact, as the AABB used in the CAM search will in fact be a cube, it is likely that fewer photons will be returned from this CAM search compared to a *k*D-tree search, resulting in reduced computation to perform during the sphere refinement. In practice, such an approach can be combined with the preferred *k*NN search procedure, as photon to search origin distance must be calculated in both routines.

6.2.2.5 PS CAM Scalability

The CAM architecture described is inherently scalable. As an AABB is used as the search volume, a minimum of a single LNS divider is needed per cell to test photons for inclusion. As this is essentially a fixed-point subtractor with a small silicon area requirement, a massively parallel arrangement of cells in a single CAM device is feasible.

As already mentioned, if there are not enough CAM cells so that each can process a single photon, multiple photons can be stored in a small RAM and these processed sequentially. This makes a CAM device with a fixed number of cells suitable for the processing of photon searches in renderings of varying complexity. However, for the best performance, sequential processing should be minimised as far as possible.

The scalability of the solution goes beyond the massively parallel arrangement of a single CAM device. To increase the number of CAM cells available for processing, multiple CAM devices can be arranged on a board, as is the case with conventional RAMs. The result read-back procedure of reading search result vectors only from cells containing included photons can be extended hierarchically, so that results are only read from cells within CAM devices containing included photons. Of course, for the most realistic of renderings using extremely large numbers of stored photons this can be scaled even further, allowing multiple boards all containing multiple CAM devices to be used.

6.2.2.6 PS CAM Integration

All cells of a photon search CAM use the same boundaries of AABB search volume. These boundaries are broadcast to all cells prior to the commencement of a search.

When a search is initiated, photon locations previously stored within the RAM in each CAM cell are tested against these boundaries to determine whether the photons lie within the search AABB. If so, the bits corresponding to these photons are set in the search result vector. Some control is required to fetch the correct photon location data from memory. This control is dependant on the number of processing elements used; as early exit strategies can be used in some configurations.

After a search is completed, the search result bit vectors are read from the CAM cells in the hierarchical fashion described. This reduces the amount of data that must be transferred. The hierarchy of search result vectors allows photon data to be relocated and used in the subsequent irradiance calculation. The double buffering of results allows the unit to perform a search while the results from a previous search are read.

As a higher level, much like the RTI pipeline, the PS CAM could either interface to a PC running the main-line renderer through a high speed link (USB 3.0 or IEEE 1394), or to a general purpose microprocessor running the main-line renderer which would allow improved interfacing. This host renderer would broadcast the search boundaries, initiate the search, and also relocate the found photons in its own memory for further use.

Another interfacing option would be to have the output of the RTI pipeline, in which an intersection point is calculated, directly feeding the input of the PS CAM. An intermediate processing step would be required to calculate the search boundaries from the intersection point, but the processing required is simple. In this scenario, the host renderer would distribute a ray to the RTI acceleration unit, and receive intersection data on RTI test completion. AABB boundaries based on this intersection data would also be distributed to the PS CAM, with photons found in the region of the intersection returned to the rendering host on completion of CAM processing. The use of LNS arithmetic in both components is a benefit in this approach.

The feasibility of this setup depends on the processing time of each of the acceleration units, and whether they could operate on different data concurrently. This should pose few problems.

6.2.3 Summary of PS Acceleration Hardware Design

We quickly determined that performing the same test on many stored photons is ideally suited to a massively parallel implementation, where multiple photons can be tested simultaneously. In such a brute-force approach, the number of photons that can be tested in parallel is directly linked to the total processing time and therefore acceleration achievable.

To maximise the amount of parallel processing units per chip, processing requirement for each must be minimal. We show that photons can be tested for inclusion in an AABB search volume using only a single LNS divider, making a massively parallel arrangement feasible.

As photon data must be stored locally to the processing elements using RAM, a CAM architecture is the obvious choice for a PS acceleration unit. This architecture combines storage with the necessary processing requirement. We show that a photon can be tested for inclusion in an AABB volume in one to six clock cycles depending on the number of dividers used. The dividers have been proven to operate at 125MHz; however we are confident that operating speed can be increased significantly in a CAM design fabricated in smaller technology.

The solution proposed is inherently scalable. An option to increase the photon capacity of a single device is to store multiple photons per CAM cell, which are processed sequentially. However, for maximum acceleration sequential processing should be minimised and replaced with parallel processing capability. We describe how an array of CAM devices can be used to increase the number of tests that can be performed simultaneously, while explaining a hierarchical method to read the search results to minimise data volumes.

There are possible issues with the use of a CAM, namely in *k*NN searching and the post search density calculation. We propose multiple solutions to both problems, with the most suitable having already being used and proven in the traditional software algorithm.

7 ACCELERATION HARDWARE IMPLEMENTATION AND RESULTS

In the previous section two designs based on LNS arithmetic were presented for the acceleration of RTI processing and photon map searching. In this section we present the results gained from successfully prototyping these designs before providing details of the synthesis of the designs as custom IC devices.

FPGA devices were used for prototyping both acceleration designs. The FPGAs available had limited resources meaning that several compromises had to be made for the prototype implementations. For both designs, however, we were able to correctly render several test models proving that the designs operated as intended. During testing, the prototype devices were used to replace the equivalent software routine, while the remainder of the algorithm was performed in software running on a host PC.

After gaining proof that the theory behind both designs was sound, various configurations of both designs were synthesised as custom ICs. Synopsis tools were used to perform the syntheses, with the ICs based on Faraday 90nm technology [103-105]. These syntheses provided operational clock speed and silicon area statistics. Using these results it is possible to provide estimates of the likely performance of the RTI and PS acceleration hardware post implementation as customs ICs. Using these estimations, we provide a comparison to the software execution times of the RTI and PS functions, when executed on our 3.0GHz Pentium 4 HT PC with 1GB of RAM running Windows XP SP2.

Finally we look at integration options, which allow us to present an indication of the possible speedups for total execution time over a software rendering when both acceleration devices are used.

7.1 RTI Acceleration Hardware Implementation and Results

In this section, prototype and custom very large scale integration (VLSI) IC implementations of the RTI acceleration design based on logarithmic arithmetic technology will be described.

A single RTI pipeline has been prototyped using FPGA devices, which allowed functional testing to take place and successfully proved the design operates as intended. This testing was performed with the prototype replacing the equivalent software function, while all other rendering steps were executed by the host PC running the main-line renderer. A description of the prototype, and results gained from this system are presented in the following section.

Following successful prototyping, several configurations of the acceleration unit, composed of varying number of pipelines, were synthesised as custom VLSI ICs in Faraday 90nm complementary metal oxide semiconductor (CMOS) technology. Speed and area statistics gained from these syntheses are presented, giving an estimate of the operating clock speeds achievable and silicon area required for implementation as a custom IC.

Using these synthesis statistics, we are able to present likely performance of a custom IC based RTI acceleration unit. We compare this to actual data gained from software renderings, presenting the acceleration achievable.

7.1.1 RTI Pipeline Prototype

The aim of prototyping the RTI pipeline was to prove that the design operates as intended. Limited funds and resources were available for prototyping this system, meaning that several compromises had to be made during prototype creation and testing. These compromises include the operating speed, the maximum number of model triangles which could be stored, RAM bandwidth for retrieving this triangle data and also the interfacing method to the host PC performing the rendering.

Additionally, due to the small size of the Spartan-3 FPGA devices used, multiple FPGA development boards were required to accommodate a single RTI pipeline. These were interfaced using simple parallel connections.

These limitations and issues were overcome, however, and several images were rendered accurately using the prototype proving correct operation of the design.

7.1.1.1 Description of Prototype Hardware

The FPGAs used for prototyping were all Xilinx Spartan-3 family, two XC3S1500-4C devices containing the pipeline and a single XC3S400-4C controlling data flow to and from the host PC [106]. All FPGAs were embedded on RaggedStone1 development boards from Enterpoint UK [107-109].

The RTI pipeline design was created in VHDL, and was synthesised using Xilinx WebPack 9.1 software. Platform flash devices on the development boards were programmed with the design using the Xilinx IMPACT application, allowing the FPGA devices to be configured on start-up.

A PCI interface was provided on the RaggedStone1 development boards; however no dedicated controller was available. To make use of this PCI interface a full PCI controller must be implemented in the FPGA device, which would take resources away from the design. For this reason, it was decided to use a slower parallel connection which would allow the FPGA devices to accommodate the design and still be tested fully.

7.1.1.2 Prototype Implementation and Operation

There were several issues encountered with the low-cost prototyping system used. In this section we present details of the issues encountered, and discuss how these issues were overcome allowing images to be rendered and correct operation to be proven.

Reworking the Design

The RTI pipeline design had to undergo some modification for prototype implementation using the available resources. The main changes were enforced due to the fact that the design for a single RTI unit was split across three distinct FPGA development boards due to the limited resources available on each of the FPGA devices. The two larger FPGA devices were used for implementation of the RTI pipeline and controller, while the smaller device was used to handle communication with the host PC.

Interfacing

A simple bi-directional parallel interface was used to communicate with the host PC. This negated the need to implement a PCI controller within the FPGA, freeing resources for the design.

Between each of the development boards, a 32-bit parallel interface was implemented. The link between the two boards containing the pipeline was unidirectional. At the particular pipeline stage when the data had to be transferred from the first of the FPGA boards to the second, the pipeline data for this stage was simply split into 32-bit words and transferred, one at a time, to the second board. This occurred during each pipeline stage.

The development board accommodating the smaller FPGA device was used to control the flow of data to and from the two boards containing the design, and also the host PC. This FPGA device had the task of reconstructing the 8-bit data values received from the PC into 32-bit words, and sending these to either of the FPGAs containing the pipeline, where these values were then stored in the attached RAM expansion modules as triangle data. This reconstruction also occurred for ray data, which was sent to the FPGA containing the start of the pipeline. Results from the RTI calculations, stored on the FPGA containing the end of the pipeline, were retrieved and split into 8-bit values before being sent back to the host PC.

Memory

There was only enough memory available on each device [106] for one copy of the LNS adder/subtractor ROMs. This was not an issue, as access to these ROMs was multiplexed between all LNS adders/subtractors on any particular FPGA.

Due to lack of storage on the FPGAs for triangle data storage, a RAM expansion module was successfully designed and manufactured, and used with each of the two FPGAs containing the pipeline [110]. Each of these modules provided 8Mb of storage and attached to the headers present on the development boards. As the data interface to these modules had a width of 32-bits, several read cycles had to be completed during every pipeline stage to retrieve the required triangle data for processing. Four words are required by the FPGA containing the start of the pipeline, and six are required by the remainder of the pipeline in the second FPGA. This means that the prototype can render models containing a maximum of 299,593 triangles, which is more than adequate for functional testing.

Implementation Summary

All changes to the design for prototype implementation, except the partitioning of the pipeline, were simply implemented using a 'wrapper'. This meant that changes to the design were minimised as far as possible. The wrapper provided controllers to handle all data operations, such as inter-board communications, multiplexed ROM reads, and triangle data reads from the RAM expansion modules. All of these data operations were performed concurrently, however the pipeline could only continue processing after all were complete. The wrapper modified the clock of the pipeline accordingly.

During testing, the number of RTIs required for renderings were minimised at far as possible by reducing the resolution of renderings and using simple models containing few triangles. This was due to the processing times for the RTI acceleration prototype being slower than the equivalent software times.

A diagram showing the various prototyping components and the data flows can be seen in Figure 44. A photograph of the RTI pipeline prototype is shown in Figure 45.

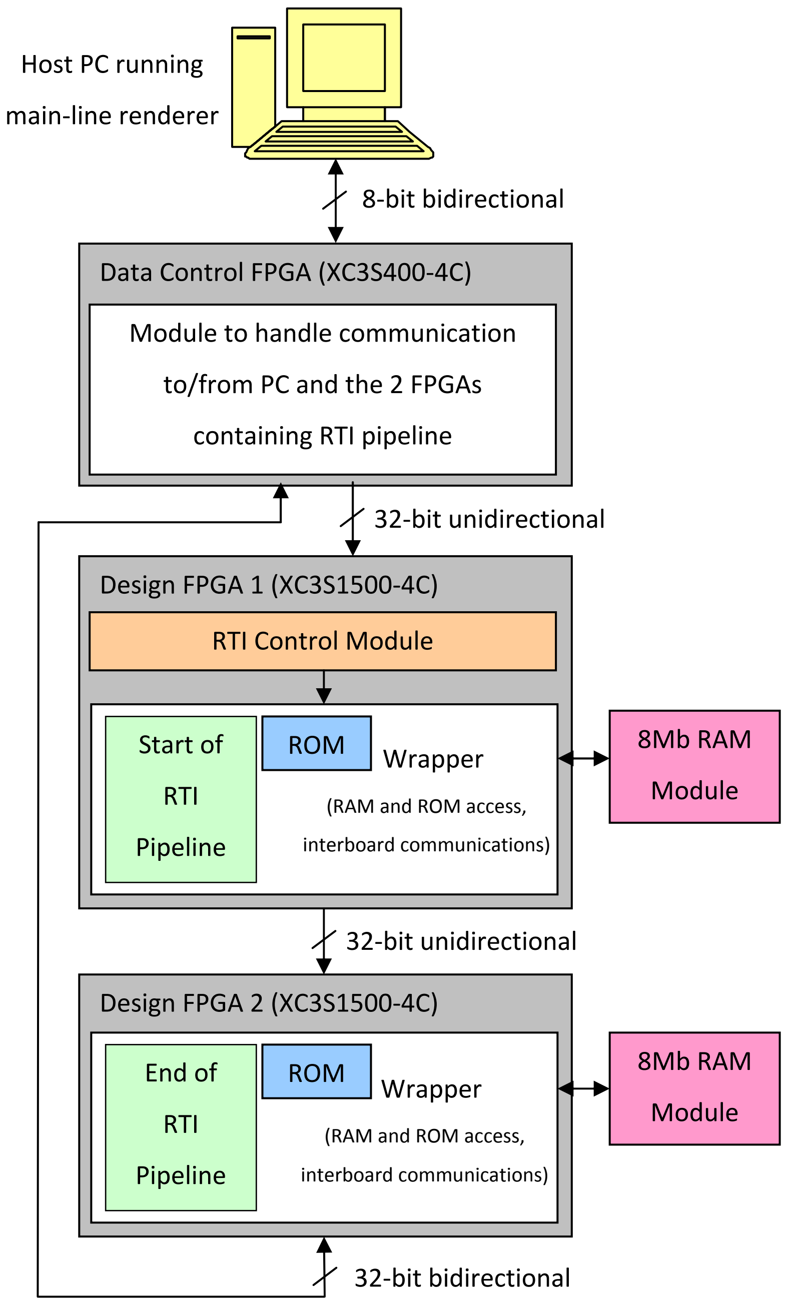


Figure 44: Final prototype design detailing components and dataflow.

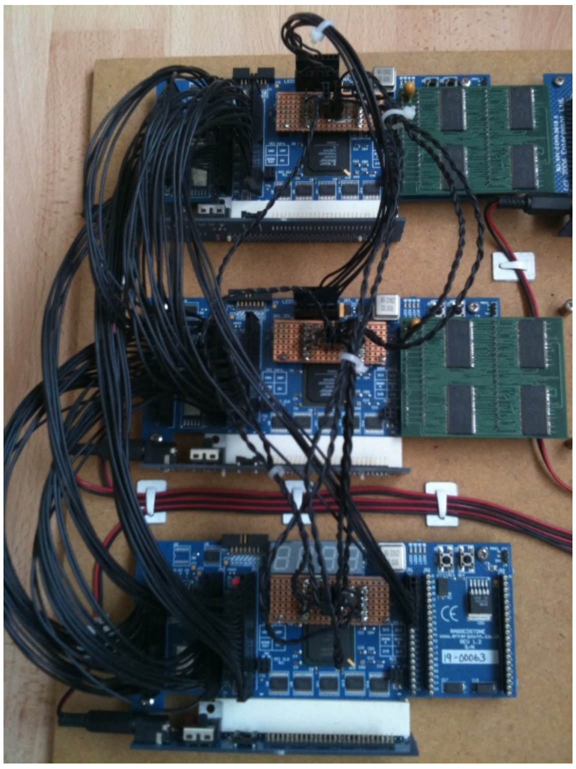


Figure 45: Photograph of prototype RTI acceleration hardware. See the attached RAM expansion modules on the development boards containing the pipeline (top and middle).

7.1.1.3 Prototype Results

We have already discussed in detail the modifications made to the RTI pipeline for prototype implementation. With optimisation, we were able to successfully run the FPGA devices containing this modified RTI pipeline design with a maximum clock speed of 66MHz. This was not the operating clock speed of the pipeline itself, which was closer to 1MHz due to the multiple ROM and RAM reads, and data transfer between FPGA boards undertaken at every pipeline stage. The actual operating speed of the RTI pipeline could not be calculated due to the variable nature of the deglitching time required in the many data transfers.

With the aim of reducing the time taken to produce renderings using the prototype, image size and model triangle count were reduced as far as possible. This minimised the numbers of rays which had to be sent to the prototype RTI unit, the number of triangles to be tested against these rays, and also the number of results to send back. This did however mean that the models rendered during testing were fairly simplistic.

Images produced while using the prototype RTI pipeline, with the remaining rendering functions performed in software, are shown in Figure 46, Figure 47 and Figure 48 below. Images in Figure 46 were rendered using simple models and photon mapping was disabled during their rendering. More complex models were rendered in Figure 47 and Figure 48 while photon mapping was enabled.

These successful renderings prove that the RTI pipeline design operates as intended.

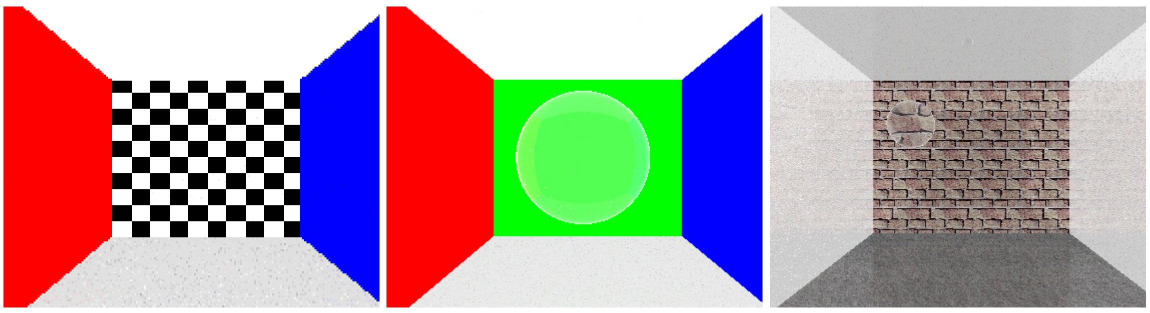


Figure 46: Images created using rendering system incorporating prototype RTI pipeline. Images show proof of correct barycentric coordinate calculation (used for texture mapping). Models contain 10, 1210 and 7590 triangles (left to right).

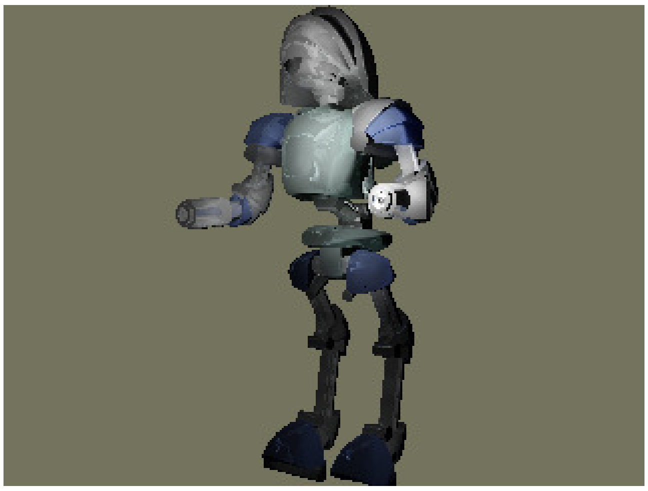


Figure 47: Image created using more complex model during prototype RTI pipeline testing. Other rendering functions were performed in software. Model contains 22,054 triangles.

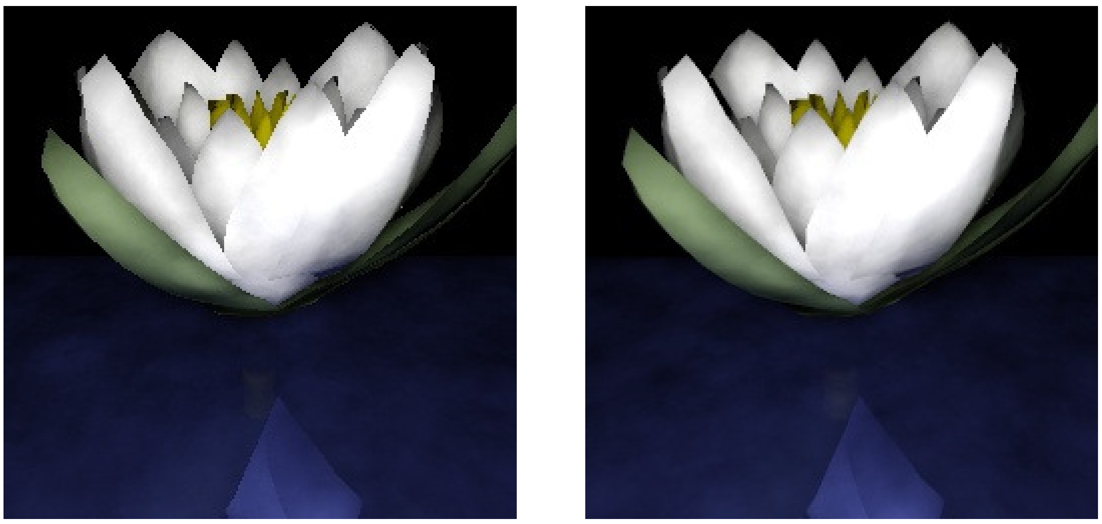


Figure 48: Left shows rendering of a Water Lily model created using prototype RTI pipeline, with other rendering functions performed in software. Right shows a wholly software rendering of the same model at a higher resolution. Model contains 1426 triangles.

7.1.2 Implementation of the RTI Acceleration Unit as a Custom VLSI IC

In this work we have simulated, synthesised and produced preliminary layouts for ICs based on the RTI acceleration unit design. No devices have been fabricated at present. In this section we present synthesis results for ICs containing varying numbers of RTI pipelines. The results presented are gained from Synopsis Design Compiler version B-2008.09-SP3, for a Faraday 90nm CMOS process.

These results, which detail the silicon area required and the possible operational clock speeds, help us to estimate the overall reduction in software ray processing time possible by utilising the RTI acceleration units. We present the possible speed-ups achievable using one to many pipelines based on the synthesis data and data gained from software renderings of two models: the Room Scene as analysed in section 3 (in various complexities), and a Water Lily model which is a refined version of the model rendered using the prototype. These test models were rendered in software using the most efficient RAS structure investigated, that of the *k*D-tree, which were built using the optimal build settings found. This ensures that fair comparison with efficient software renderings can be made.

7.1.2.1 Synthesis Results

The area statistics gained from the synthesis of an IC containing a single RTI pipeline are shown in Table 18. RAM, used to store the triangle data, is not included in these results.

Synthesised Design	**Size (mm^2)**
RTI Pipeline (No ROMs)	0.710
RTI Pipeline (including ROMS)	3.488
Full RTI IC, including ROMs, I/O control and pipeline control	3.505

Table 18: Synthesised areas of RTI pipeline components.

One of the parallelisation options looked at was using multiple RTI units to process the same ray. This meant that each unit would only need to access a subset of all scene triangles, thus reducing the per-RTI pipeline RAM requirement.

ICs based on this idea containing 4 and 8 pipelines have been synthesised, with two storage capacities for each configuration. This means that the ICs have a triangle capacity of between 4K and 32K triangles, for 4 pipelines each having capacity for 1K triangles and 8 pipelines each having capacity for 4K triangles respectively. The intention is that an array of these ICs would be used majority of renderings.

Synthesis results for these ICs are shown in Table 19. The total sizes include all required ROMs and control logic. Note that to achieve the required depth in the 4K RAMs, multiple narrow RAMs had to be used side by side due to the chosen technology. This had the impact of increasing total RAM area for these configurations due to the overheads associated with small RAM devices. This was not the case with the 1K deep RAMS.

RTI Pipelines	Triangle Capacity per Pipeline	Triangle Capacity Total	Total RAM Size (mm^2)	Total IC Size (mm^2)
4	1024	4096	2.642	16.083
8	1024	8192	5.285	32.060
4	4096	16384	14.664	25.531
8	4096	32768	29.328	50.944

Table 19: Synthesised areas of multiple parallel RTI units for processing the same ray including on-chip RAM

7.1.2.1.1 Reduction of Bit-Width

It may be the case that the range and precision offered by 32-bit arithmetic may not be necessary throughout the entire RTI calculation. For example, it may only be required up to and including the ray-plane intersection calculation where lack of precision often causes problems.

This would mean that a 20-bit implementation, for example, could be used for the remainder of the processing in the pipeline. The transition from 32-bit arithmetic to 20-bit would be an easy one, as both data formats would maintain the sign bit and 8-bit integer part, while the fraction would be truncated from 23-bits to 11.

Such a reduction in bit width could potentially reduce the overall area of the design. Area savings would be especially prominent in the case of LNS adders and subtractors, where the size of the required lookup tables grow exponentially with bit-width.

No experiments have been undertaken to verify that such a scheme would in fact operate correctly, so no further detail is provided in this work. Further work will however look at finding areas of the pipeline where the range and precision offered by 32-bit arithmetic is not fully utilised, and implementing such a bit-reduction technique if appropriate.

7.1.2.2 Timing Results

Results from all synthesised RTI acceleration designs indicated that an implementation created using Faraday 90nm technology would be able to run at clock speeds in excess of 200MHz. In the results presented in the remainder of this work, we have assumed the use of a clock speed of 200MHz which was shown to be achievable. We use this operating speed, along with statistics gained from software renderings to calculate likely speedups for ray processing time over our software implementation. Comparisons are made with our rendering software using the most efficient RAS investigated, which was the *k*D-tree (which was optimally built).

The results presented are based on a brute-force approach, where the RTI acceleration does not make use of any RAS technique. This means that all rays must be tested for intersections with all model triangles. The hardware acceleration units must therefore process many more RTI calculations than undertaken in the software renderings. As no RAS is used, ray processing time consists of RTI processing time only. For fair comparisons with software renderings, we make use of the equivalent ray processing time, which is the sum of software RTI and RAS processing time.

We assume that, for the vast majority of renderings, arrays of several ICs each containing multiple RTI pipelines will be used. This will make the 1K and 4K triangle capacity pipelines already synthesised suitable for rendering most models. The triangle capacity has no impact on timings, and therefore there is no timing penalty if the pipeline triangle capacity was increased.

For reference, useful statistics from our software renderings to which comparisons are made are shown in Table 20. Four of these models are those used in our software analysis (the Room Scene with varying levels of complexity), and the other is a Water Lily model which is a refined version of the model rendered using the prototype RTI acceleration unit. A software rendering of the Water Lily model is shown in Figure 49.

Property	Room Scene (1)	Room Scene (2)	Room Scene (3)	Room Scene (4)	Water Lily
RTI Time (s)	14.606	18.197	22.929	25.383	9.268
RAS Time (s)	8.407	11.547	12.917	15.168	5.277
Total Ray Processing Time (s)	23.013	29.744	35.846	40.551	14.545
Time to process a ray (µs)	7.905	9.272	10.386	11.036	9.356
Total Rays Fired	2911071	3208094	3451328	3674338	1554355
Model Triangles	54841	104381	151976	207583	6146

Table 20: Relevant information gained from software renderings of test models.

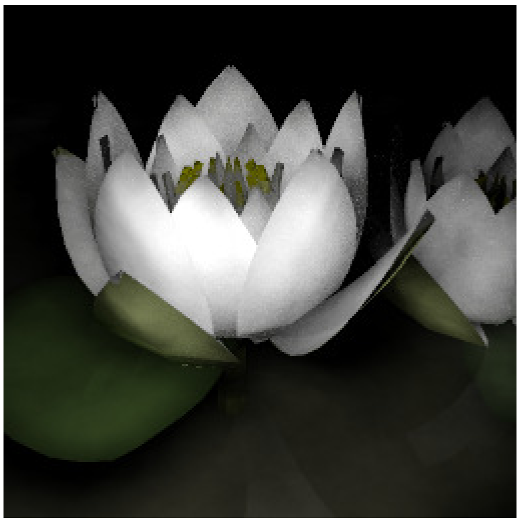


Figure 49: Software rendering of Water Lily model. Model contains 6146 triangles.

A normal system would have multiple pipelines, however, for completeness we begin our analysis by looking at timing results for a single pipeline in the rendering of the test models. We then move on to look at a more typical scenario where the use of arrays of multiple ICs, each containing multiple pipelines, is assumed.

For a single RTI pipeline running at 200MHz we can estimate the total ray processing time when a brute-force approach is taken. These processing times are shown in Table 21, and represent a worst case situation, where the pipeline must fully flush after the processing of every ray. Note the large number of RTI calculations to perform, due to the fact that every triangle must be tested for intersection with each ray.

Reference Rendering	Total Rays	Total RTIs	Total Clock Cycles	Worst Case Processing Time (s)	Data Transfer Time (s)	Total RTI Time (s)
Room Scene (1)	2911071	$1.596\text{x}10^{11}$	$1.597\text{x}10^{11}$	798.725	0.466	799.191
Room Scene (2)	3208094	$3.349\text{x}10^{11}$	$3.350\text{x}10^{11}$	1674.866	0.513	1675.379
Room Scene (3)	3451328	$5.245\text{x}10^{11}$	$5.246\text{x}10^{11}$	2623.182	0.552	2623.734
Room Scene (4)	3674338	$7.627\text{x}10^{11}$	$7.629\text{x}10^{11}$	3817.275	0.588	3817.863
Water Lily	1554355	$9.553\ \text{x}10^{9}$	$9.553\ \text{x}10^{9}$	47.765	0.249	48.014

Table 21: Projected worst case RTI processing and data transfer time using single RTI pipeline.

The above results also include the time taken to transfer ray data to the pipeline and read the results for the first intersection. A maximum of eight 32-bit words of ray data and triangle specifiers must be transmitted to the RTI pipeline before processing can take place. A similar amount of data must be transferred back to the rendering system after processing has completed, provided that a valid RTI has been found which we assume is the case for each ray. As stated while looking at the RTI unit design, we envisage that the pipeline, or multiple pipelines, will be connected to a host PC using a high speed interface such as IEEE 1394 or USB 3.0, running at say

3.2Gbit/s. We use this interface speed to calculate the total RTI processing and data transfer time for a single pipeline.

We can see that using a single pipeline operating at 200MHz, total RTI processing time using no RAS scales with the number of RTIs to perform, which is to be expected. While the total processing times are much larger than those in software, we are not taking into account the differences in clock speed of the two systems. Looking at the rendering of the Room Scene (1) model, we can see that there is a difference in total ray processing time of 34.727 times, however there is also a 15 times difference in clock speeds, from 3.0GHz for the software rendering to 200MHz in the RTI pipeline. Taking this into account, we could suggest that if both systems had equal clock speeds then a single RTI pipeline would take just over twice as long as the software in ray processing tasks, despite having to process many orders of magnitude more RTI calculations (approximately 2000 time more for Room Scene (1), and 3820 times more on average for all models).

These results for a single pipeline are very encouraging; however one of the major benefits of the RTI acceleration pipeline design is its scalability. We will now look at what acceleration is possible using multiple pipelines in an array of ICs.

Reference Rendering	Number of ICs	Number of RTI Pipelines	Worst Case Processing Time (s)	Data Transfer Time (s)	Total Ray Processing Time (s)	Times Speedup (over single pipeline)	Times Speedup (over software)
Room Scene (1)	1	8	100.286	0.509	100.796	7.929	**0.228**
	4	32	25.443	0.539	25.981	30.761	**0.886**
	8	64	12.969	0.553	13.522	59.103	**1.702**
	16	128	6.739	0.568	7.307	109.373	**3.150**
	32	256	3.624	0.582	4.206	190.012	**5.471**
	64	512	2.067	0.597	2.664	299.997	**8.640**
Room Scene (2)	1	8	209.841	0.561	210.403	7.963	**0.141**
	4	32	52.869	0.593	53.463	31.337	**0.556**
	8	64	26.707	0.610	27.317	61.331	**1.089**
	16	128	13.634	0.626	14.260	117.488	**2.086**
	32	256	7.090	0.642	7.732	216.681	**3.847**
	64	512	3.818	0.658	4.475	374.386	**6.646**
Room Scene (3)	1	8	328.411	0.604	329.015	7.975	**0.109**
	4	32	82.556	0.638	83.194	31.538	**0.431**
	8	64	41.571	0.656	42.227	62.134	**0.849**
	16	128	21.088	0.673	21.761	120.570	**1.647**
	32	256	10.837	0.690	11.527	227.616	**3.110**
	64	512	5.712	0.708	6.419	408.745	**5.584**
Room Scene (4)	1	8	477.333	0.643	477.976	3.992	**0.085**
	4	32	119.802	0.680	120.482	15.837	**0.337**
	8	64	60.222	0.698	60.921	31.320	**0.666**
	16	128	30.424	0.716	31.140	61.274	**1.302**
	32	256	15.524	0.735	16.259	117.354	**2.494**
	64	512	8.084	0.753	8.837	215.918	**4.589**
Water Lily	1	8	6.241	0.272	6.513	7.413	**2.233**
	4	32	1.764	0.288	2.052	23.527	**7.089**
	8	64	1.018	0.295	1.313	36.769	**11.074**
	16	128	0.645	0.303	0.948	50.926	**15.340**
	32	256	0.459	0.311	0.769	62.780	**18.904**
	64	512	0.365	0.319	0.684	70.582	**21.267**

Table 22: Projected worst case RTI processing and data transfer times for arrays of between 1 and 64 ICs each containing 8 RTI pipelines. Speedups are in comparison to total software ray processing time.

Table 22 shows total processing time for arrays of 1 to 64 ICs, each containing 8 RTI pipelines. These times relate to each pipeline concurrently processing the same ray. Again we present worst case processing time and data transfer time, which now includes calculating the closest intersection to the ray origin from all pipelines. This imposes little overhead, as the distance checks can be simply cascaded using a tree of LNS dividers to find the smallest intersection distance. The times speedup for each configuration is calculated using the total ray processing time in software (RTI + RAS), as the hardware processing time is the equivalent due to no RAS technique being employed.

We can see that to achieve any speedup in ray processing over our software implementation, a minimum of 64 to 128 pipelines would be required. Again, the two reasons behind this are the differences in clock speed between the PC running the software implementation and the hardware pipelines, and the fact that many more RTI calculations must be performed by the pipelines. If the two systems were able to operate with equal clock speeds, then approximately 4 to 8 pipelines would achieve acceleration over the software implementation for the room scene models, and fewer for the water lily model, despite the dramatic increase in RTI calculations performed in the pipelines.

As the differences in clock speed do exist, the array of pipelines required to meet or exceed the performance of the software implementation can be realised through the use of few ICs, each containing several pipelines. Table 22 shows that there is a near linear relationship in total processing time between a single and multiple pipelines, with the small non-linear difference attributable to the determination of the minimum intersection distance from all pipelines. Such calculations will always be required, whether they are performed on-chip with large numbers of pipelines per IC, or both on and off chip, using an array of medium sized ICs, as assumed in the results presented. The ideal ratio of pipelines per IC/number of ICs needs further investigation, and will be looked at in our future work.

All results presented so far have been for the worst case, where all pipelines used must fully flush before the processing of a new ray can commence. We earlier discussed relatively simple ways to avoid the need to flush the pipeline between the processing of separate rays (see section 6.1.3.5). None of these features have been included in the design or assumed during the calculation of the timing results presented. If these features were implemented, RTI pipeline utilisation would increase leading to a reduction in total RTI processing time.

7.1.2.2.1 Smart Operation

The main premise of the RTI acceleration design was scalability, meaning that RAS techniques commonly used in software were not needed. The design, however, is general enough to also operate in a smart mode. In such a situation, a RAS would be used to reduce the number of RTIs that must be computed per ray.

The use of RAS techniques with the RTI acceleration hardware is in the initial stages of investigation. There is some promise in such an approach, however, due to the reduction in the number of RTIs to process, which could lead to reductions in ray processing time or hardware requirement. Further investigation on this matter will form the basis of some of our future work.

For initial analysis we have looked at directly replacing the software RTI function with our RTI acceleration unit, and maintaining the use of the same *k*D-trees and RAS traversal functions as used in our software renderings. Details of how *k*D-tree nodes would map to triangles in the RTI pipelines needs further investigation. There are many approaches for the many different RAS techniques, such as storing triangles in different *k*D-tree leaves or grid cells in different pipeline memories so that rays can be tested for intersections with these voxels/cells concurrently. However, here we have simply assumed that the triangles in each leaf node of the *k*D-tree are evenly distributed amongst all RTI pipelines, and all are stored in pipeline memories starting at the same memory address.

To test a ray for intersection with triangles in any *k*D-tree leaf voxel would then entail supplying the starting address of this voxels triangles, and the maximum number of triangles stored in any one of the pipeline memories, *m*. Triangle data from this starting address would then tested for intersections with the ray, which is constant for all pipelines. The memory address would be incremented on each clock cycle, meaning that after *m* clock cycles, all triangles within this *k*D-tree leaf node will have been tested for intersections with the ray. For optimal performance in this case, the maximum number of triangles in any leaf node should match the number of pipelines available, making $m = 1$.

As, in software, the RTI function is called multiple times per ray (for every leaf node intersected), we have assumed a simple change to the RTI unit design and main-line renderer that would allow more efficient operation. This assumed design change would allow the starting address and number of triangles contained in multiple leaf nodes, those that a ray could intersect, to be sent to the RTI pipeline controllers when a new ray is processed. This would allow multiple leaf nodes of the *k*D-tree to be tested consecutively in the pipelines against this ray, meaning that the pipelines would not need to flush between the processing of the triangles in each leaf.

Early timing estimates based on this suggestion, for 1 to 16 RTI pipelines, are shown in Table 23. We have used statistics gained from our software renderings in the calculation of these estimates, as shown in Table 24. In particular, the average number of intersected leaf nodes per ray has been taken as the number of starting addresses and leaf triangle counts that would be sent to the RTI acceleration unit on the commencement of processing a ray.

Standard Rendering	Number of RTI Pipelines	Processing Time (s)	Data Transfer Time (s)	Total RTI Time (s)	Times Speedup (RTI Only, over software)	Times Speedup (Total Ray Processing, over single pipeline)	Times Speedup (Total Ray Processing, over software)
Room Scene (1)	1	1.150	0.735	1.886	7.745	1.000	**2.236**
	2	0.823	0.750	1.573	9.288	1.031	**2.306**
	4	0.659	0.764	1.423	10.263	1.047	**2.341**
	8	0.577	0.779	1.356	10.773	1.054	**2.357**
	16	0.577	0.794	1.370	10.658	1.053	**2.354**
Room Scene (2)	1	1.437	0.895	2.332	7.802	1.000	**2.143**
	2	0.991	0.911	1.902	9.565	1.032	**2.212**
	4	0.768	0.927	1.696	10.732	1.048	**2.246**
	8	0.657	0.943	1.600	11.373	1.056	**2.262**
	16	0.657	0.959	1.616	11.260	1.054	**2.260**
Room Scene (3)	1	1.578	0.979	2.556	8.969	1.000	**2.317**
	2	1.082	0.996	2.078	11.033	1.032	**2.390**
	4	0.834	1.013	1.848	12.409	1.048	**2.428**
	8	0.711	1.030	1.741	13.169	1.056	**2.445**
	16	0.711	1.048	1.758	13.040	1.054	**2.443**
Room Scene (4)	1	1.830	1.117	2.948	8.612	1.000	**2.238**
	2	1.227	1.136	2.363	10.742	1.033	**2.313**
	4	0.926	1.154	2.080	12.203	1.050	**2.351**
	8	0.775	1.172	1.948	13.032	1.058	**2.369**
	16	0.775	1.191	1.966	12.911	1.057	**2.367**
Water Lily	1	0.850	0.355	1.205	7.690	1.000	**2.244**
	2	0.574	0.363	0.937	9.887	1.043	**2.341**
	4	0.436	0.371	0.807	11.479	1.065	**2.391**
	8	0.368	0.379	0.746	12.419	1.076	**2.415**
	16	0.333	0.386	0.720	12.879	1.081	**2.426**

Table 23: Estimated average case processing times for 1 to 16 RTI pipelines when operating with *k*D-tree RAS. Data transfer time includes sending a batch of leaves to process at once.

Property	Room Scene (1)	Room Scene (2)	Room Scene (3)	Room Scene (4)	Water Lily
Average Leaf Nodes Intersected per Ray	5.630	6.950	7.179	8.203	4.432
Average Triangles per Leaf Node	7.590	7.381	7.328	7.383	16.246

Table 24: *k*D-tree RAS traversal statistics gained from reference renderings.

Table 23 shows both speedups achievable over RTI processing only and for ray processing as a whole. We can see that an order-of-magnitude speedup in RTI processing time is achievable with a much reduced hardware requirement; however RAS node traversal has not been accelerated meaning that the speedup over total ray processing time is reduced.

We can also see that increasing the number of pipelines available for the processing of RTI calculations has a much reduced impact than in the brute force scenario. This is due to several factors including the non-accelerated RAS functions taking a large proportion of the overall ray-processing time and the fact that pipeline flush cycles account for a greater proportion of total processing time when a RAS is employed. Of course, the techniques mentioned earlier which avoid the need for flushing the pipelines between the processing of rays would still apply when the pipelines are used in this configuration, and as there should be fewer triangles to process per ray, these techniques could be of particular benefit in this scenario.

If the processing associated with RAS traversal could be accelerated to similar levels as the RTI processing, an order of magnitude speedup in overall ray processing time would be attainable. Acceleration of the RAS traversal functions will form part of the future work undertaken to investigate RAS usage with the RTI acceleration unit.

7.1.2.3 Summary of RTI Acceleration Unit VLSI Synthesis

We have presented results from the synthesis of ICs containing one, four and eight RTI pipelines. These results show that we could expect an operating frequency of 200MHz if implemented in Faraday 90nm technology. Area results are also presented, for a single pipeline this shows that the majority of IC area is utilised by the ROMs required in the LNS addition and subtraction elements. Area requirements for ICs containing four and eight parallel RTI units for processing the same ray were also shown. These included the required RAM to store between 4K and 32K triangles, and were within the limits of a medium sized IC.

Using the results from IC synthesis we were able to project the reductions in ray processing time possible over the equivalent software routines. In these projections and subsequent speedup calculations, we used the actual data gained from the software renderings of our test models which use *k*D-trees as RAS, which are constructed using the most efficient settings found.

In a brute-force approach, the RTI acceleration hardware had to perform on average 3820 times more RTI calculations than were performed during the execution of the equivalent software function. This meant that to achieve any acceleration over the software renderings, which were executed on a microprocessor running at more than 15 times the operating frequency, the scalability of the solution had to be utilised. A highly scaled RTI acceleration hardware implementation operating at 200MHz, using between 8 and 512 pipelines in 1 to 64 ICs was used to estimate speedups over the total software ray processing time. It was shown that significant speedups were achievable, over an order of magnitude for the Water Lily model, and that further increases would be possible by increasing the number of pipelines.

The RTI acceleration unit is also flexible enough to operate using a RAS technique, in a smart mode. Although this suggestion has not been fully investigated, preliminary estimations indicate that the volume of hardware required to achieve substantial acceleration over the software RTI function is drastically reduced in comparison to a brute-force approach. We show that as little as 2 pipelines on a single IC could offer order of magnitude speedups over RTI processing time. Acceleration over the total software ray processing time is however reduced in comparison, as the RAS traversal function has not yet been optimised.

In these initial estimations we assumed that the same *k*D-trees as employed in our software renderings were used and that the RTI unit directly replaced the equivalent software function. We also assumed that data on the leaf nodes to be tested for intersection with any particular ray could be sent to the RTI unit in one transaction. Details on such usage scenarios, and also triangle/containing volume distribution between pipelines, are yet to be fully considered. This will be looked at as part of our future work concerning integration and optimisation of RAS techniques and traversal.

7.2 PS Acceleration Hardware Implementation and Results

In this section we present details of the implementation of the photon search CAM design. The design has been prototyped using an FPGA device allowing functional testing to take place.

Following a successful prototyping stage, the design was synthesised as an IC based on Faraday 90nm technology. Speed and area statistics gained from the syntheses of several CAM configurations are presented, showing operating clock speeds achievable and silicon area required for a custom IC implementation.

Using these synthesis statistics, we are able to present likely performance of the CAM in its varying configurations. For this, we use actual data gained from the software renderings of our test models, allowing us to compare time taken for a custom IC photon search to our software implementation.

7.2.1 PS CAM Prototype

The same resources were available which were used for the RTI pipeline prototype, meaning that several of the same issues were encountered. This again meant that we had to make compromises during prototype implementation and testing, which included the speed of operation, the maximum number of photons which could be stored, RAM bandwidth and also the interfacing method to the host PC.

As well as this, due to the size of the FPGA used, only a single logarithmic divider could be contained for use as the comparison element in each CAM cell. This was only one of four possible combinations.

The post search irradiance calculation was performed in software. For all renderings shown, the photons found that were outside of a sphere equal in radius to the search radius were discarded. This allowed the area of a circle to be used in the density calculation. This is our preferred approach, as discussed in section 6.2.2.4.

7.2.1.1 Description of Prototype Hardware

A single Xilinx Spartan-3 family XC3S1500-4C FPGA device [106] embedded on a RaggedStone1 development board from Enterpoint UK was used for CAM prototyping [107-109]. Like the RTI unit, the design was created in VHDL, and was synthesised and programmed to the platform flash of the development board using Xilinx WebPack software.

We again decided against implementing a controller for the PCI interface on the development board, meaning that more resources were available for the CAM design. Instead, we decided to use the same parallel interface that was created during RTI pipeline prototyping.

The prototype was created to prove that the CAM design operated correctly. With this in mind we aimed to maximise the number of photons which could be stored, and therefore searched. The more photons which can be stored in the photon map means, in general, the more realistic the model for rendering can be.

7.2.1.2 Prototype Implementation and Operation

Like the RTI acceleration unit prototype, the design of the photon search CAM had to be altered to make the most of the limited prototyping hardware.

We have already discussed different types and configurations of memory that could be used in the PS CAM. In the prototype, the internal Block RAMs of the Spartan-3 were used, as any external memory module could not offer the required throughput. The Spartan-3 has 32 Block RAMs, each 18Kbits in size [111]. Each Block RAM can be addressed separately, but if the required RAM size is greater than 18Kbits, then multiple RAMs are combined and this functionality is lost. Therefore, the maximum number CAM cells achievable using this device, where each cell has its own independently addressable memory, is 32. Each of these RAMs were used to store photon location data and a unique photon identifier, which allowed for initial testing of our post search photon relocation strategy as described in section 6.2.2.2 in particular.

Unfortunately, even with only a single LNS divider per CAM cell and logic to control memory access, early exits and input/output, the FPGA resources available would not allow 32 CAM cells to be implemented. With optimisation, the maximum number of cells achievable was 27, with almost 100% utilisation of FPGA resources. The RAM in each of these cells could store 146 photons, giving a total capacity of 3942.

The structure of the PS CAM prototype is shown in Figure 50. As this shows, for communication between the host PC running the main-line renderer and the prototype CAM, the same 8-bit bi-directional asynchronous parallel interface as used in the RTI acceleration hardware was implemented. This interface was used to send photon positions and identities to the CAM during initial configuration, as well as search boundaries and results between the CAM and rendering host before and after searches. Logic was required to reassemble the 8-bit input into 32-bit words used for search boundary data, and 18-bit values suitable for storage in the block RAM devices.

For transmission of the search results, the bit vector representing whether a CAM cell has at least one included photon was split into bytes and sent over the interface. Then, for cells containing an included photon, bit vectors representing the inclusion of photons within this cell were split and transferred. This was done, in order, for all cells allowing for the relocation of the included photons in software.

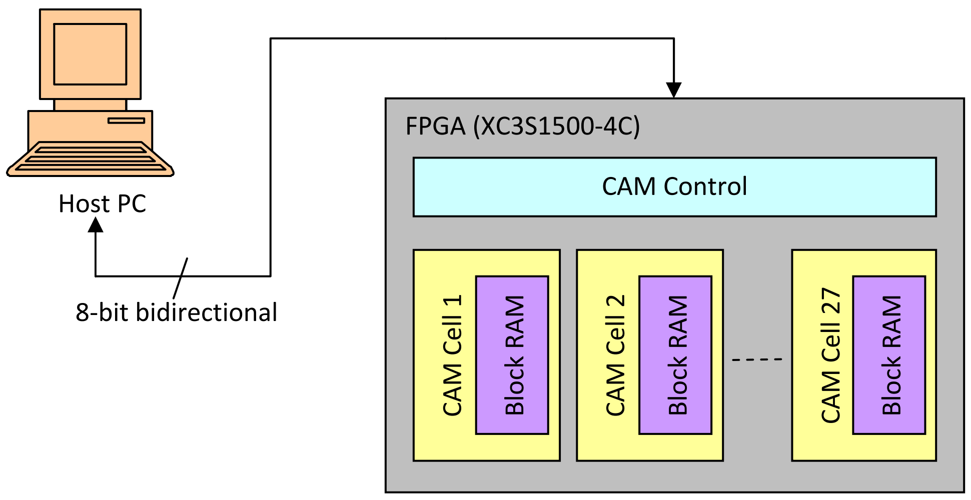


Figure 50: Final CAM prototype design detailing various components.

Although the PS CAM prototype allowed the theory of operation to be proven, due to the lack of photon storage available it is not suited to the rendering of complex models. We present an idea in the following section that enabled us to render moderately complex models using our prototype.

7.2.1.2.1 Reducing Noise using Multiple Photon Searches

A maximum of 3942 stored photons is not sufficient to render any realistically complex models. One solution which we have used with the prototype is to combine several rendered images, each of which is noisy due to being generated using fewer than the optimal number of photons. As photons are fired and stored randomly with each photon firing pass, the noise within the images is also random.

To reduce this to reasonable levels, we have found that averaging several of these noisy images works well. We acknowledge that, as photon mapping is a biased rendering algorithm, averaging several images does not converge to a correct result; however, we have found that visually acceptable results for most model surfaces can be produced using this method.

This technique has allowed us to render more complex models than we first thought possible using the prototype photon search CAM.

7.2.1.3 Prototype Results

Like the RTI acceleration hardware prototype, various modifications were made to the design for prototype implementation. This modified design was optimised to operate at 66MHz. In this section we present renderings generated using the prototype CAM proving that theory behind the photon search CAM search is correct.

The complexity of the model involved has little impact on the number of photon searches which must be performed. The number of photon searches which must be undertaken is however proportional to the number of rays fired. For this reason, the resolution of the renderings produced was reduced as far as possible. This minimised the number of rays, and therefore search boundaries and results to transfer during the rendering, thus reducing hardware execution and overall rendering time.

The rendering of a simple model containing only spheres is shown in Figure 51. This simple model does not need a large amount of photons to be stored for the generation of noise free images. This meant that the multiple-pass method described in the previous section was not needed.

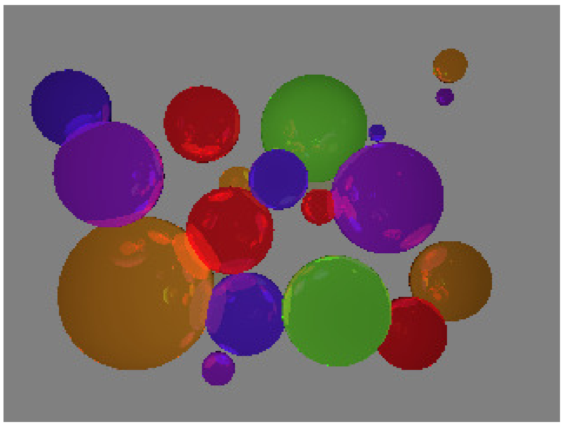


Figure 51: Rendering of simple model containing only coloured spheres using prototype photon search CAM. Approximately 3600 photons were stored, and no averaging was used on this image.

The Room Scene model used for our analysis of the photon mapping algorithm was also rendered. This model is however moderately complex, and could not be rendered without noise using the number of photons that could be stored in the prototype. For this reason, the multiple rendering pass technique was used to reduce noise when rendering this model. The image produced using 10 rendering passes, each using the prototype PS CAM, is shown in Figure 52. A software rendering of the same model, at a higher resolution, is also shown for comparison.

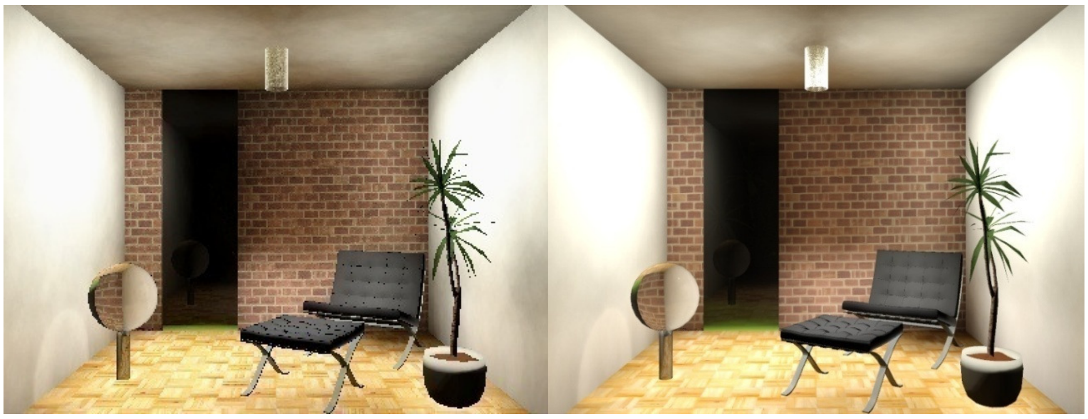


Figure 52: Left shows rendering of test model using prototype photon search CAM. This was formed by averaging 10 images, where the CAM stored approximately 3800 photons in each case. Right shows software rendering of the same model at a higher resolution. Note the difference in the renderings of the light shade, due to the averaging used with the biased photon mapping algorithm.

7.2.2 Implementation of the Photon Search CAM as a Custom VLSI IC

We have simulated and synthesised ICs based on the PS acceleration unit design, however, like the RTI design, no devices have yet been fabricated. We present results of the synthesis of the photon search CAM design as a custom IC in this section. The statistics presented are gained from Synopsis Design Compiler version B-2008.09-SP3, for a Faraday 90nm CMOS process.

These results, which detail the likely operational clock speeds, help us to estimate the overall reduction in photon search times achievable by using the CAM. As there are several options in terms of number of CAM cells, number of LNS dividers per cell, as well as width and depth of RAM devices, we present results of what we perceive to be some of the more promising configurations.

For this comparison, we use actual data gained from software renderings of our test models, allowing us to compare time taken for a photon search in hardware to our software implementation. The settings used in the software renderings were those just sufficient to give a realistic output in a reasonable execution time. Of course, both the search radius and the number of photons emitted can be lowered to reduce the photon search time in software, however the quality of output will be diminished. The software renderings we use ensure a fair comparison for a realistic rendering.

For a fair base for comparison between the various CAM configurations, all ICs considered were capable of storing and searching 32K photons. Due to the scalability of the design, however, we envisage that arrays of the CAM ICs will be used for the renderings of most models, much the same as the RTI acceleration hardware and indeed conventional memories, giving much increased overall capacity.

IC synthesis also reveals silicon area requirement. We look the trends in terms of silicon area for the 32K capacity devices, and also synthesise two of the possible CAM configurations as custom ICs capable of storing and searching 128K photons to see how area requirement scales. We will also discuss and show details of one possible method to reduce the silicon area requirement.

7.2.2.1 Synthesis Results

In this section we present the results of synthesis of the photon search CAM design as custom ICs in various configurations. For this synthesis, we used the Faraday 90nm CMOS technology libraries [103]. For the RAM devices required, the compatible synchronous high-density single-port SRAMs from the Faraday UMC 90nm library [112] were used.

IC Synthesis of the photon search CAM design indicated that an implementation created using Faraday 90nm technology would be able to operate at clock speeds in excess of 350MHz. An achievable operating clock speed of 333MHz is assumed in the calculation of execution times of various configurations of the PS CAM in the following section.

Turning to silicon area requirement, synthesised areas for a 32K photon search IC in various configurations are shown in Table 25. We present configurations for minimal area, where only a single LNS divider is used to perform the needed magnitude comparisons, and also for fastest processing where six dividers are used. Note that when six dividers are used, photon location data is tested simultaneously in all axes, meaning a 96-bit RAM interface is required.

Cells	Photons/Cell	RAM Depth	No of RAMs	LNS Dividers	RAM Area(mm^2)	Area (mm^2)
16	2048	6144	16	1	5.963	8.922
32	1024	3072	32	1	7.452	10.628
64	512	1536	64	1	8.601	12.154
128	256	768	128	1	11.107	15.405
256	128	384	256	1	15.899	21.642
16	2048	2048	16	6	5.990	9.185
32	1024	1024	32	6	7.414	11.056
64	512	512	64	6	10.185	14.678
128	256	256	128	6	15.803	21.993

Table 25: Area synthesis results for 32K photon search CAM in various configurations.

Figure 53 and Figure 54 show how the RAM and total area requirement varies for the different CAM configurations. As the number of cells increase, the number of RAMs required also increases. To maintain a capacity of 32K photons, the size of each of these RAMs decreases. The overhead associated with the use of small RAM devices leads to an overall rise in total IC area. There is also a small increase not associated with the RAMs, which is due to the processing and control within the increased numbers of cells, and also the extra logic required to control the increased cell count. Therefore, the use of fewer deep RAMs would be preferential in terms of silicon area to a greater number of smaller devices.

Unfortunately, having few deep RAMs is not optimal for performance. This leads to fewer CAM cells for parallel processing, each with deeper RAMs meaning that more sequential processing of photons must take place. This lengthens the processing time of all cells, and therefore the CAM as a whole.

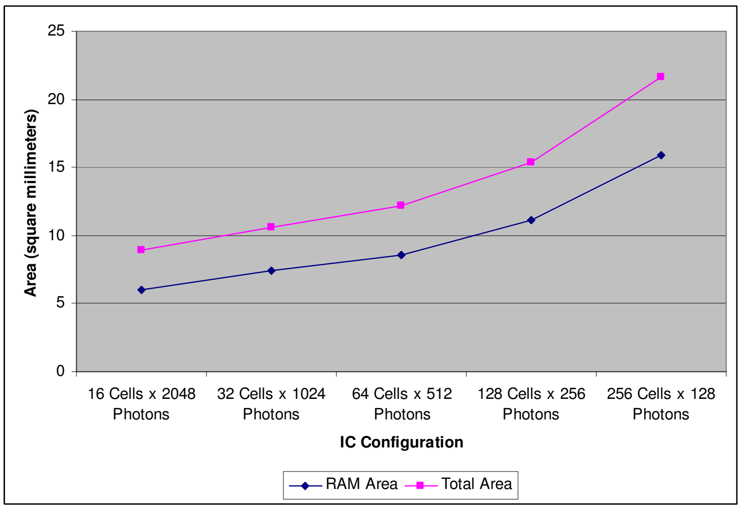


Figure 53: Total synthesised area and RAM area for 32K photon search ICs in various configurations. All configurations use 32-bit wide RAMs and a single LNS divider.

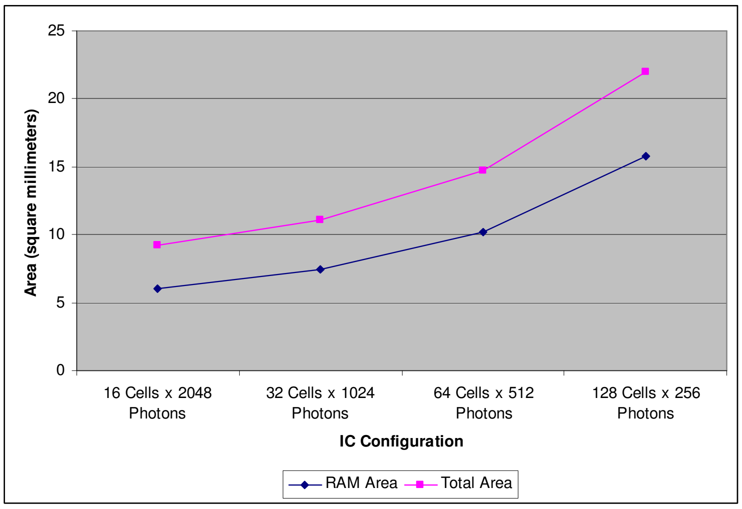


Figure 54: Total synthesised area and RAM area for 32K photon search ICs in various configurations. All configurations use 96-bit wide RAMs and 6 LNS dividers.

To ascertain how the total and RAM area scale with photon capacity, two configurations of the CAM were modified resulting in ICs that could store and search 128K photons. The configurations of these ICs and the area statistics gained from their syntheses are shown in Table 26.

Cells	Photons/Cell	RAM Interface (bits)	LNS Dividers	RAM Area (mm^2)	Area (mm^2)
128	1024	32	1	29.809	37.198
128	1024	96	6	29.657	39.052

Table 26: Synthesised area of photon search CAM IC capable of storing 128K photons.

We can see from these results that there is a linear increase in RAM area, as we would expect as we are using a greater number of the same RAM devices. The overall area also exhibits an almost linear increase; with the areas for the 128K photon search ICs being just less than four times those of the equivalent 32K devices. This is a large area requirement for a device capable of storing only a moderate number of photons. We therefore looked for a way of reducing this area while keeping all of the benefits of the CAM.

7.2.2.1.1 Reducing PS CAM Silicon Area

Where six LNS dividers are used, there is no opportunity for early exiting as all magnitude comparisons are performed simultaneously. Therefore, the requirement for individual addressing access to a RAM, which is useful for early exit strategies, is no longer present. This means that we could simply stream photon locations (in three dimensions) to CAM cells at a rate of one photon per clock cycle, which the CAM cells would process at the same rate. A large, wide RAM, shared between cells could be used for this purpose. This would be advantageous, as we have previously seen that there is an area overhead associated with the use of multiple small RAM devices.

A block diagram of this modified design is shown in Figure 55. Note that little control is required in each cell, as the RAM is addressed by the CAM controller which also handles inputs and search result outputs.

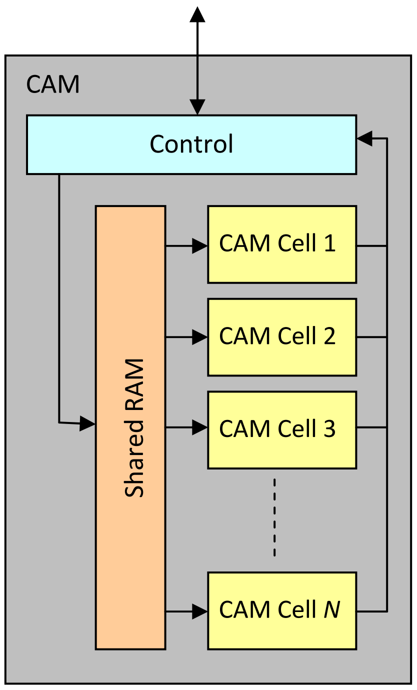


Figure 55: Block diagram of photon search CAM modified to use a large RAM shared between cells.

We have synthesised a 128K photon search CAM as a custom IC based on this idea to observe the difference in area requirements. For the shared RAM, we used 96 side-by-side RAMs of 128 bits by 1024 words, the maximum width available for the depth required for our chosen devices. The results of this synthesis are shown in Table 27. Note the reduction in RAM and total areas, which could possibly be reduced further if larger RAM devices are made available.

Cells	Photons/Cell	RAM Interface	LNS Dividers	RAM Area (mm^2)	Area (mm^2)
128	1024	12288 bits shared	6 per cell	24.602	28.467

Table 27: Synthesis results for 128K photon search CAM using large RAM shared between all cells.

It should be noted that this idea of using a shared memory between all cells could also be used in the case where a single LNS divider and a 32-bit RAM interface per cell is used. However, no early exiting is possible in this configuration, thus the processing of each photon will consistently take the maximum of six clock cycles to complete.

7.2.2.2 Timing Results

As already stated, IC Synthesis of the photon search CAM design indicated that an implementation created using Faraday 90nm technology would be able to operate at clock speeds in excess of 350MHz. This operating speed was indicated for all configurations of the CAM. In the results presented in the remainder of this work we have based our results on the use of a clock speed of 333MHz, which was shown to be achievable.

Estimates are provided for the photon search processing time using synthesised devices for the rendering of two test models, one of which is based on the Room Scene analysed in section 3, and the other the Water Lily model shown in Figure 49. In contrast to the RTI comparisons, we will use only one variation of the Room Scene model due to the reduced impact model complexity has on photon search processing.

These processing times are compared with the equivalent from our software renderer. In these calculations, we use the actual number of photon searches performed and the number of photons stored, which we gained from our software rendering profiles. This data is shown for reference in Table 28.

Property	Room Scene	Water Lily
Total PS Time (s)	89.961	20.962
Single PS Time (μs)	39.211	19.896
Total Photons Stored	65756	24565
Total Photon Searches	2294307	1053596
Average Photons Found/Search	85.686	86.617

Table 28: Relevant information gained from software renderings of test models.

As the synthesised ICs have capacity to store 32K photons, an array of these devices must be used in the rendering of the Room Scene, where 65,756 photons are stored. The CAM design used in the acceleration hardware is highly scalable, and we foresee that banks of CAM ICs would be used in the majority of renderings, although of course larger capacity CAMs can be constructed within limits of a medium to large size IC, as already shown.

For the rendering of the Room Scene, a minimum of three 32K PS CAM ICs are required, giving total photon capacity of 96K. The Water Lily model, with only 24,565 photons stored, can utilize as little as a single IC. Photon search processing times for the two models using various CAM configurations within these ICs are shown in Table 29. For results where a single LNS divider is used, worst case figures are presented. This would only be the case if every photon tested was within the search volume during a photon search, making early exits impossible.

We envisage that the PS acceleration hardware will be connected to a host PC using one or more high speed interfaces such as IEEE 1394 or USB 3.0, each operating at say 3.2Gbit/s. We can use this interface speed to calculate the AABB distribution and result transmission times. The total time to perform a photon search is the maximum of the processing time and data transmission time, as the photon search results are buffered (using a simple double buffer arrangement), allowing a new search to be performed during the transmission of the results of the previous search.

Profiling results from our software renderings allowed us to calculate the average number of photons found per photon search, as shown in Table 28. We use these figures in the calculation of the data transfer times, shown in Table 29. These are worse case times, where every one of the found photons resides in a different CAM cell. This means that the full results from each of these CAM cells must be read.

Reference Rendering	ICs	LNS Dividers	CAM Cells	Total CAM Cells	Max Photons Stored per Cell	WC Processing Cycles per Search	WC Processing Time per Search (µs)	WC Processing Time Total (s)	WC Data Transfer Time per Search (µs)	WC Data Transfer Time Total (s)	Times Speedup
Room Scene	3	1	32	96	685	4110	12.342	28.317	18.433	42.291	2.127
	3	1	128	384	172	1032	3.099	7.110	4.787	10.982	8.192
	3	6	32	96	685	685	2.057	4.720	18.433	42.291	2.127
	3	6	128	384	172	172	0.517	1.185	4.787	10.982	8.192
Water Lily	1	1	32	96	768	4608	12.342	14.579	20.879	21.998	0.953
	1	1	128	384	192	1152	3.099	3.645	5.378	5.666	3.699
	1	6	32	96	768	768	2.057	2.430	20.879	21.998	0.953
	1	6	128	384	192	192	5.165	0.607	5.378	5.666	3.699

Table 29: Photon search processing times for renderings of test models for varying number of LNS dividers and cells per CAM. Room Scene results are for the use of 3 ICs to give the required storage capacity, whereas the Water Lily uses the minimum of 1 IC.

While a respectable reduction in photon search time is possible using the minimum number of ICs capable of storing the required number of photons, a major selling point of the CAM design is it scalability. We envisage that arrays of ICs, each containing many CAM cells, will be used in the majority of renderings. Table 30 shows timing results for 4 to 128 ICs each containing 128 CAM cells, with each of these cells using six LNS dividers. Due to the increase in the number of cells, fewer photons are stored per cell leading to a reduction in the sequential processing required. Worst case data transfer results are presented in Table 31 for both a single and two of the 3.2Gbit/s connections to the host renderer for these configurations. Note that worst case data transfer times exhibit a parabolic trend depending on the number of cells available. For a small number of cells, there are a large number of photon result search vectors that must be read. For a large number of cells, there is a reduced volume of search result bit vectors (as not all cells will contain photons within the AABB); however there is an increased upfront volume to read for all of the ICs and CAM cells.

Reference Rendering	ICs	Total CAM Cells	Max Photons Stored per Cell	WC Processing Cycles per Search	WC Processing Time per Search (µs)	WC Processing Time Total (s)
Room Scene	4	512	129	129	0.387	0.889
	8	1024	65	65	0.195	0.448
	16	2048	33	33	0.099	0.227
	32	4096	17	17	0.051	0.117
	64	8192	9	9	0.027	0.062
	128	16384	5	5	0.015	0.034
Water Lily	4	512	48	48	0.144	0.152
	8	1024	24	24	0.072	0.076
	16	2048	12	12	0.036	0.038
	32	4096	6	6	0.018	0.019
	64	8192	3	3	0.009	0.009
	128	16384	2	2	0.006	0.006

Table 30: Photon search processing times for renderings of test models using varying number of ICs, each containing 128 CAM cells with 6 LNS dividers per cell.

		WC Data Transfer using single 3.2Gbit link			WC Data Transfer using two 3.2Gbit links			BC Data Transfer using single 3.2Gbit link		
Reference Rendering	ICs	Transfer Time per Search (µs)	Transfer Time Total (s)	Times Speedup	Transfer Time per Search (µs)	Transfer Time Total (s)	Times Speedup	Transfer Time per Search (µs)	Transfer Time Total (s)	Times Speedup
Room Scene	4	3.675	8.433	**10.668**	1.838	4.216	**21.336**	0.142	0.325	**101.218**
	8	2.123	4.871	**18.470**	1.062	2.435	**36.939**	0.143	0.328	**200.879**
	16	1.589	3.645	**24.682**	0.794	1.822	**49.364**	0.146	0.334	**288.445**
	32	1.805	4.142	**21.721**	0.903	2.071	**43.442**	0.141	0.323	**276.374**
	64	2.881	6.610	**13.610**	1.441	3.305	**27.220**	0.147	0.336	**264.712**
	128	3.661	8.400	**10.709**	1.831	4.200	**21.419**	0.165	0.379	**233.223**
Water Lily	4	1.521	1.602	**13.085**	0.760	0.801	**26.170**	0.131	0.138	**138.026**
	8	1.032	1.087	**19.276**	0.516	0.544	**38.553**	0.133	0.140	**150.156**
	16	1.030	1.085	**19.320**	0.515	0.543	**38.639**	0.135	0.142	**147.375**
	32	1.512	1.593	**13.155**	0.756	0.797	**26.310**	0.138	0.146	**144.041**
	64	2.721	2.867	**7.311**	1.361	1.434	**14.623**	0.147	0.155	**135.172**
	128	3.619	3.813	**5.498**	1.809	1.906	**10.996**	0.168	0.176	**118.780**

Table 31: Best and worst case data transfer times for rendering setups shown in Table 30. Speedups presented are for total execution time (max of processing or data transfer).

In the results presented here, for a fairly small number of overall CAM cells, we show that order-of-magnitude speedups in photon search time are possible in the worst case. Of course, the PS CAM can be scaled further, even to the extent that racks of boards each containing many ICs could be used in the rendering of large and complex models where there are many stored photons.

All results show that actual photon search computation can be preformed very quickly indeed; however the time taken for transferring the AABB boundaries to the CAM and reading back the search results dominates overall processing time in the worst case, where every found photon was assumed to be stored within a different CAM cell.

This worst case scenario is unlikely in reality. As the direction of photons emitted from light sources is random, the locations of consecutive photons stored in the photon map, or CAM, also show a high degree of randomness. For a photon search at any particular location, the probability of finding multiple photons in the same cell or finding multiple photons all in different cells will therefore be equal. In all subsequent calculations, however, worst case times are assumed.

One method of reducing the data transfer time, as already discussed, would be to store spatially close photons in the same CAM cell. This would lead to a reduction in the number of CAM cells that results must be read from after a search, and therefore decrease data transmission time. This method of photon storage has not been used in any of the testing undertaken; however the allocation of photons to cells based on spatial locality is one area where future work will be focused. If such a scheme was implemented, we could expect to see data transfer times and speedups close to those shown in the best case section of Table 31. The best case data transfer times do not exhibit the same parabolic trend as the worst case equivalents, and are fairly consistent for differing numbers of CAM cells. This is due to data being read for the minimal number of cells in all cases.

7.2.2.3 Summary of PS CAM VLSI Synthesis

In the previous sections we have presented results from IC synthesis of varying PS CAM configurations. These results showed that all of these configurations could operate at 350MHz if implemented in Faraday 90nm CMOS technology, however we used the achievable 333MHz operating clock frequency in all calculations.

Synthesis results also revealed area statistics for ICs based on various CAM configurations, all capable of storing 32K photons. As many models will require more than this number of photons, we envisage that arrays of multiple of these ICs will be used, as is the case with conventional memories. Of course, ICs of larger capacity could also be used. We synthesised two of the CAM configurations as ICs capable of storing and processing 128K photons. We showed that there was a linear increase in RAM area with an increase in photon storage, which led to a near linear rise in overall IC area.

To reduce this area, we looked at removing the many small RAM modules from within the CAM cells, and replacing them with one large shared memory. An IC was synthesised containing this modified CAM design which was again capable of storing and processing 128K photons. This arrangement helped to reduce RAM area due a reduction of area overheads associated with small RAM devices, however early exit strategies are not applicable when this approach is used as cells are unable to specify the address of the data they require.

Returning to timing results, using the results gained from the IC syntheses, we were able to project the reductions in photon search time possible over the equivalent software routine when using the minimal number of ICs. In these projections, we used the actual data gained from software renderings of test models. Results were also presented for the case where the solution was scaled to use 4 to 128 ICs, each containing 128 CAM cells, which leads to a reduction in the sequential processing required. Worst case photon search processing times showed between 1.43 times speedup for a single IC with 32 CAM cells and a single divider per cell for the Water Lily model, and 2645.91 times speedup for 128 ICs each using 128 CAM cells per IC and six dividers per cell for the Room Scene model, when compared to the respective software photon search processing times.

Of course, the design is highly scalable, and these processing times will reduce with an increase in the number of CAM cells available for parallel processing. This scalability also makes it likely that any advantage gained over software implementations can be maintained as processor speeds improve.

However, when data transmission times are taken into account the overall speedups over a software photon search are reduced. In all cases we presented the worst case figures, where photons found are all located in different CAM cells. For the Room Scene model, figures show that using 3 ICs each containing 32 cells with one LNS divider per cell the overall speedup is 2.13 times that of the software rendering, increasing to 24.68 times when 16 ICs containing 128 cells with 6 LNS dividers are used. Speedups can be improved by increasing the bandwidth available for data transfer. For the same 16 ICs with 128 cells setup, but using two 3.2Gbit/s connections, a 49.36 times speedup is achievable.

These results assume the use of a double buffering solution to avoid having to wait for results to be returned before beginning a new photon search. Such a solution was implemented in the synthesised designs.

As the data transmission times were shown to dominate PS processing time, one point of focus in future work will be the allocation of photons to CAM cells based on spatial locality. As already discussed, if one photon is found to be within a search AABB, photons close to this are also likely to be within the AABB. If these included photons are all stored in few CAM cells, this reduces the amount of search result data to be read. We presented best case results for if such a scheme was implemented, showing that a two order of magnitude speedup over a software photon search should be achievable.

7.3 Combined Acceleration Results

In the previous sections we have presented results gained from synthesis of custom VLSI ICs based on the RTI pipeline and photon search CAM designs. Both silicon areas and achievable operational clock speeds were revealed, the latter being used with statistics gained from software renderings of our test models to estimate the likely performance of these IC implementations. This has allowed us to present the speedups possible over the equivalent routines of our software renderer.

We have shown that both systems independently can offer significant speedups over their software equivalents in the worst case. The photon search function was shown to take the majority of software render time, and was therefore the priority for acceleration. Timing results presented for the CAM design show that one to two orders of magnitude speedup is possible over the equivalent software function in the worst case.

The ray processing functions account for less of the overall rendering time in software, and so did not require the same level of acceleration to achieve similar execution times to the photon search design. Results presented show that up to an order of magnitude speedup is achievable over total software ray processing time.

So far we have not looked at the combined acceleration achievable over the equivalent software functions when both devices are used, and the effect this has on overall rendering time. As the two devices work simultaneously, the effective time is dominated by the larger of the two. Estimated overall worst case render times when both acceleration devices are used are presented in this section.

In order to make these projections, we have assumed that the time taken to execute the ancillary main-line rendering functions in software remains constant, and is not dependant on whether photon searching and RTI processing is performed in software or hardware. One further assumption is that one or more high-speed interfaces such as IEEE 1394 or USB 3.0 can be used, and are able to offer throughput of 3.2Gbit/s each.

All results presented are for the use of the acceleration hardware shown in Table 32, and are worst case figures. For software rendering comparison, we will again make use of software renderings of the Room Scene (in one complexity), as analysed in section 3, and also the Water Lily model as shown in Figure 49.

Results presented are for the RTI acceleration hardware operating in a brute-force mode, meaning there is no processing associated with RAS traversal. The 64 or 32 RTI ICs assumed have a total capacity of 512K or 256K triangles respectively, with each IC having a triangle capacity of 8K (1K per pipeline) and size of approximately 32mm^2. Using this hardware, we could reasonably expect significant speedups to be maintained as model complexity increases from the 55K in the Room Scene model presented in this section, to say large models containing 100K or 150K triangles.

Resources	Room Scene	Water Lily
RTI	64 ICs each containing 8 RTI pipelines	32 ICs each containing 8 RTI pipelines
RAS	N/A	N/A
PS	16 ICs each with 128 cells	8 ICs each with 128 cells

Table 32: Assumed acceleration hardware resources.

During our discussions on the integration of the two acceleration hardware devices, two main options were proposed. The first was that of both devices interfacing directly to the main-line renderer. The second was that of the RTI acceleration unit interfacing directly to the PS acceleration unit, so that when an intersection was found a photon search could be initiated making use of the calculated intersection coordinates. Illustrations of both configurations are shown in Figure 56.

In both cases photon searching and RTI computation can execute simultaneously. Such a usage scenario can be taken into account in total hardware execution time, which will be dominated by the unit having the longest overall execution time.

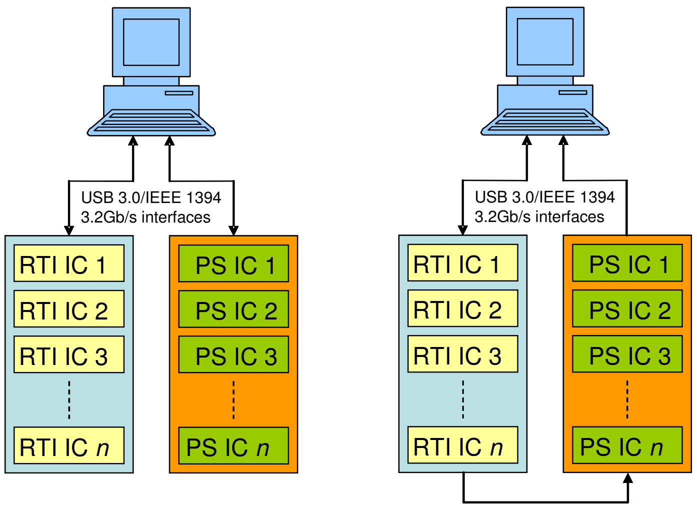


Figure 56: Rendering setup using host PC running the rendering using multiple RTI and PS acceleration units over high speed interfaces. Left shows both devices interfacing directly to host renderer. Right shown RTI unit forwarding results to PS unit.

We begin by looking at the combined acceleration achieved over the optimised functions only. Table 33 shows total combined hardware processing time for the RTI and PS acceleration units alongside the equivalent time taken in the comparison software renderings. In these calculations we have assumed the use of two 3.2Gbit/s links to the host renderer for the PS acceleration unit, and a single link for the RTI unit. The total software processing times presented include the time taken in the ray processing routines (RTI and RAS), and also PS.

		Software Processing Time (s)	Hardware Processing Time (s)	Times Speedup
Room Scene	RTI Time (s)	14.606	2.664	
	RAS Time (s)	8.407	N/A	
	PS Time (s)	89.961	1.822 *Included in RTI*	
	Total Time (s)	112.974	2.664	**42.408**
Water Lily	RTI Time (s)	9.268	0.769	
	RAS Time (s)	5.277	N/A	
	PS Time (s)	20.962	0.544 *Included in RTI*	
	Total Time (s)	35.507	0.769	**46.173**

Table 33: Combined speedup of accelerated functions only for overlapping execution of hardware units.

While a 42 times speedup is achievable over the core functions in software, we must now look at the acceleration achievable in total rendering time, when the non-optimised main-line rendering functions are taken into account. Speedups achievable in overall rendering time are shown in Table 34 and Figure 57.

		Software Only		Estimated PS and RTI times in hardware, main-line in software		Times Speedup
		Time (s)	% of Total	Time (s)	% of Total	
Room Scene	RTI Time (s)	14.606	11.819	2.664	20.081	
	RAS Time (s)	8.407	6.803	N/A	N/A	
	PS Time (s)	89.961	72.798	*Included in RTI*	*Included in RTI*	
	Other (s)	10.602	8.579	10.602	79.919	
	Total Time (s)	123.576	100.000	13.266	100.000	**9.315**
Water Lily	RTI Time (s)	9.268	23.367	0.769	15.617	
	RAS Time (s)	5.277	13.305	N/A	N/A	
	PS Time (s)	20.962	52.852	*Included in RTI*	*Included in RTI*	
	Other (s)	4.155	10.476	4.155	84.383	
	Total Time (s)	39.662	100.000	4.924	100.000	**8.055**

Table 34: Total estimated rendering time if PS and RTI acceleration hardware is used and their processing occurs simultaneously. No RAS is employed with RTI unit, i.e. operating in brute-force mode.

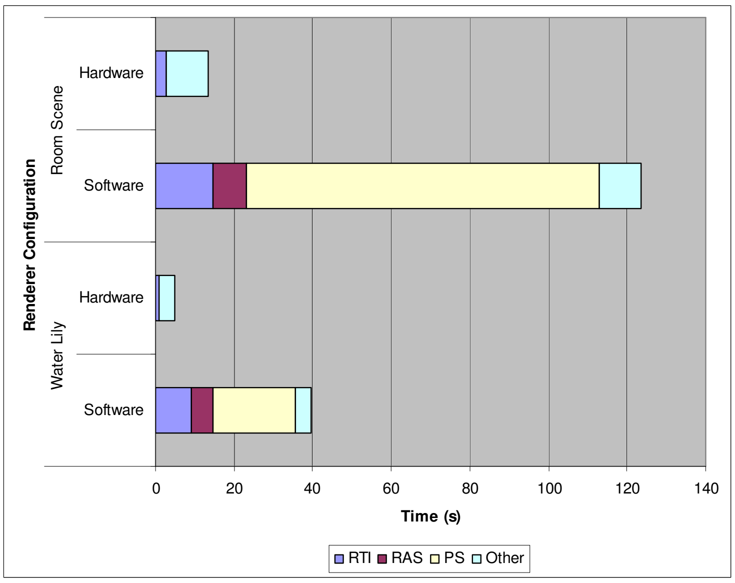


Figure 57: Graphical representation of estimated rendering time if PS and RTI acceleration hardware is used and their processing can overlap. Shown alongside software only rendering for comparison. PS processing time is absorbed in RTI time for hardware setup.

For both models the speedup achievable is close to one order-of-magnitude, and this could be increased, to a limit, through the addition of further hardware due to the scalability of both acceleration designs. Importantly, this advantage can be maintained with increasing processor speeds.

For example, in the unlikely case that processor speeds were increased to the extent that total software processing time decreased by a factor of 2, using the same number of hardware acceleration devices would result in only a small reduction in overall speedup to approximately 8 times for the Room Scene model, and 7 times for the Water Lily model. By increasing the number of acceleration devices available for use, the 9.3 and 8 times speedup figures for the two models can easily be reinstated.

As the results show, with the reductions in ray processing and photon searching times, the time spent in other rendering functions now dominates overall rendering time. In the estimated rendering times for both models, the time spent in main-line rendering tasks consumed approximately 80% of total execution time when both hardware acceleration devices are used. For significant further reductions in overall rendering time, these ancillary main-line tasks must be accelerated. The integration of the RTI and PS acceleration units with a rendering controller will form a substantial component of our future efforts. Optimisation of the remaining rendering functions will be investigated during this work.

7.3.1 Smart Operation

In section 7.1.2.2.1 we presented preliminary ideas on using the RTI acceleration unit in a smart mode, making use of RAS techniques. Early estimations showed that fewer RTI pipelines would be required to achieve significant acceleration over the software RTI function due to the reduced number of RTI calculations that must be processed. These estimations were based on the use of the same *k*D-trees as our software renderings with the hardware directly replacing the software function.

As this approach has shown some early promise, we present estimations of the speedups achievable for total execution time when the RTI and PS acceleration units are used, and the RTI unit makes use of the same *k*D-trees as employed in our software renderings. We again assume that the PS and RTI units can operate simultaneously. Another assumption made is that the *k*D-tree traversal time, like the main-line processing time, remains constant.

Table 35 and Figure 58 show the effect on overall rendering time of using a single RTI IC containing 8 RTI pipelines in a smart mode. The PS hardware assumed are 16 and 8 ICs each containing 128 CAM cells for the Room Scene and Water Lily models respectively. These results show that significant acceleration is achievable; however this is reduced in comparison to a brute-force approach. This is due to the non-accelerated RAS traversal and ancillary main-line rendering functions, which consume approximately 90% of total rendering time.

The use of RAS techniques with the RTI acceleration units will be considered further in our future work.

		Software Only		Estimated PS and RTI times in hardware, main-line in software		Times Speedup
		Time (s)	% of Total	Time (s)	% of Total	
Room Scene	**RTI Time (s)**	14.606	11.819	1.356	6.658	
	RAS Time (s)	8.407	6.803	8.407	41.282	
	PS Time (s)	89.961	72.798	*Included in RTI/RAS*	*Included in RTI/RAS*	
	Other (s)	10.602	8.579	10.602	52.060	
	Total Time (s)	123.576	100.000	20.365	100.000	**6.068**
Water Lily	**RTI Time (s)**	9.268	23.367	0.746	7.330	
	RAS Time (s)	5.277	13.305	5.277	51.847	
	PS Time (s)	20.962	52.852	*Included in RTI/RAS*	*Included in RTI/RAS*	
	Other (s)	4.155	10.476	4.155	40.823	
	Total Time (s)	39.662	100.000	10.178	100.000	**3.897**

Table 35: Total estimated rendering time if PS and RTI acceleration hardware is used and their processing occurs simultaneously. Smart operation is assumed with *k*D-tree RAS employed with RTI unit.

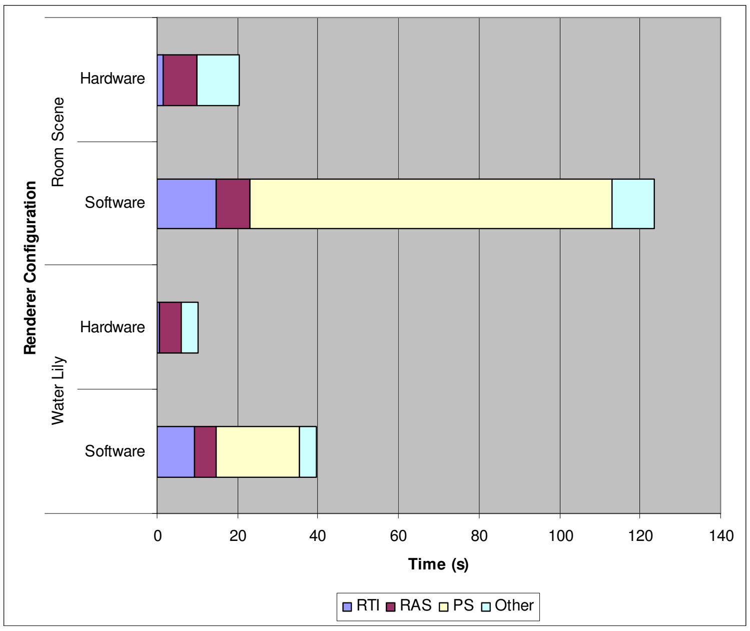


Figure 58: Graphical representation of estimated rendering time if PS and RTI acceleration hardware is used and their processing can overlap. Shown alongside software only rendering for comparison. PS computation is absorbed in RTI and RAS time for hardware setup. Smart operation is assumed with *k*D-tree RAS employed with RTI unit.

7.3.2 Summary of Combined Acceleration Results

In the previous section we presented results for the combined acceleration of the optimised functions, and also showed the impact that these speedups have on overall rendering time.

Looking only at the accelerated functions, we show that a worst case 42 times speedup over software processing time is achievable. This figure is based on the speedups provided by both the photon search acceleration unit and the RTI acceleration hardware, with both devices operating simultaneously. As the ray processing functions consume a smaller percentage of overall software rendering time, they require less acceleration to achieve the same runtimes as the photon search hardware. Due to the anticipated simultaneous execution, total processing time of the acceleration hardware is the maximum time taken by any one of the devices.

This 42 times speedup, however, does not take into account the various ancillary main-line rendering tasks that have not yet been optimised. If these are taken into account, we have shown that close to order of magnitude speedups in overall rendering time would be possible in the worst case.

With the reduction in ray processing and PS time, main-line processing time now consumes a much larger proportion of overall render time. While further RTI and PS hardware can be used to reduce the time taken for ray processing and PS processing to a certain limit, for a more significant reduction in overall rendering time the acceleration of the main-line rendering functions must be investigated.

For the main-line functions, due to the various different tasks performed, it is unlikely that a fixed function hardware unit like the RTI or PS acceleration hardware would be appropriate. It is more likely that a micro-coded unit would be required, in which case an LNS arithmetic platform would be preferable to ease integration with the existing acceleration devices. Further benefits may also be possible in this situation, in acceleration of the main-line functions due to the LNS arithmetic processing and also in the use of custom interfaces between the various acceleration units. These interfaces could be designed specifically for the purpose and possibly offer greater throughput than the IEEE 1394 or USB 3.0 links assumed in the existing designs. Again, such a setup will be investigated in our future work. A basic diagram of such a setup is shown in Figure 59.

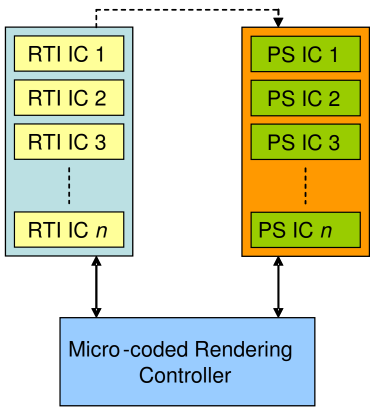


Figure 59: Rendering setup using custom micro-coded controller to manage the rendering, which makes use of multiple RTI and PS acceleration units using custom high speed interfaces.

We also presented preliminary results based on using the RTI acceleration unit in a smart mode, making use of a RAS technique. This had the effect of reducing the number of RTI pipelines required to achieve significant acceleration over the software RTI function. However, acceleration for overall ray processing time was reduced in comparison to a brute-force approach, due to the non-accelerated RAS traversal software function which consumes a large proportion of overall rendering time.

The major benefit in the use of a RAS is the reduced number of RTIs to process, leading to a reduction in the number of RTI pipelines required. Due to the advantages that could be possible, investigating RAS integration with the RTI unit, and RAS traversal acceleration, will be considered in future work.

7.4 Summary of Acceleration Hardware Implementation and Results

We have provided details and results for FPGA based prototype implementations of both RTI and PS acceleration hardware designs. The prototypes allowed us to prove that the theory behind our designs was correct, and were used in the rendering of several test images. The prototype devices were used to replace the equivalent functions in software.

Following successful prototyping, various configurations of both designs, containing different numbers of RTI pipelines and PS CAM cells, were synthesised as custom ICs in 90nm technology. The synthesised designs were based on 32-bit LNS arithmetic elements, proven to have greater accuracy than their floating-point equivalents in additions and subtractions, and no error in multiplications and divisions. While no images were produced using a complete prototype (both RTI and PS computation in hardware for the same rendering), the arithmetic elements used are intrinsically more accurate that their floating-point counterparts meaning no degradation in image quality would result from the use of the acceleration devices. These syntheses resulted in probable IC areas and operating speeds for these various configurations of the acceleration units.

Area results for multiple RTI pipeline ICs with each pipeline having sufficient capacity for 1K or 4K triangles, and PS CAM ICs capable of holding 32K photon locations showed that these were within the limits of a medium sized chip. For example, an IC containing 8 RTI pipelines, each of which have the capacity to store 1K triangles, has a silicon area of approximately 32mm^2. Of course, 1K triangles per RTI pipeline could easily be reduced to minimise IC area, while a solution of using a large shared RAM between all CAM cells was presented to reduce the area requirement of a CAM implementation.

Using the achievable operating frequency revealed by IC synthesis, as well as statistics from software renderings of our test models, we were able to project likely speedups for the two rendering functions being accelerated by these devices. We were also able to present possible speedups for the photon mapping algorithm as a whole, when both acceleration devices are used.

For the RTI unit operating in a brute-force approach, the results show that an array of custom ICs, each containing 8 RTI pipelines operating at 200MHz could provide up to an order of magnitude speedup over the software ray processing time in the worst case, when each pipeline must be flushed after the processing of a ray. However, ideas to avoid having to flush the pipeline were discussed which could lead to further speedups. It is likely that this level of performance can be maintained as model complexity rises, to a limit of say 100K or 150K triangles, which is well within the triangle limit of a 32 IC array of RTI units each containing 8 pipelines with 1K of triangle storage.

We also looked at a preliminary idea for use of the RTI acceleration hardware in a smart approach, using the same *k*D-trees as used in our software renderings. Over one order of magnitude speedup over software RTI processing time was achieved, however when taking into account the non-optimised RAS traversal processing, the speedup for overall ray processing time was reduced.

As for the photon search CAM, synthesis results show that an operating speed of more than 350MHz was achievable, however, we based our results on a CAM capable of storing 32K photons running at 333MHz. Timing results showed that order of magnitude speedups were achievable using multiple ICs with 32K photon capacity in 128 CAM cells. Of course, due to the scalable nature of both acceleration designs, processing times can be reduced if more hardware is used for parallel processing which will lead to greater speedups over software implementations.

We finally looked at the impact of using both acceleration devices on rendering time. These calculations were based on the use of 64 or 32 RTI ICs each containing 8 pipelines interfaced to a rendering host by a single 3.2Gbit/s link, and either 8 or 16 of the 32K capacity PS CAM ICs using two of these connections.

Looking first at the acceleration achieved over the optimised functions only, we show that a 42 times speedup is achievable. This assumes that both the RTI and PS hardware can process simultaneously, and that overall hardware execution time is the maximum of the two individual execution times. The software execution time used in comparison is the sum of RTI, RAS and PS execution times.

The same assumptions were made in the calculation of speedups over total software rendering time, and in addition we assumed that the main-line rendering time remained unchanged. It was shown that very close to one order of magnitude speedup is possible. It was shown that the reductions in the ray processing and photon search execution times lead to the vast majority of execution time (80%) being consumed in the ancillary main-line rendering tasks.

With two of the three core rendering functions accelerated, and the other eliminated (when the RTI unit is used in a brute-force approach), we have shown that the main-line rendering functions now account for a much larger proportion of total algorithm execution time. For significant further acceleration, the ancillary rendering tasks performed in the main-line rendering functions must also be accelerated. One proposal is to use an LNS based micro-coded processing unit for executing the main-line rendering tasks. This could conceivably provide acceleration for these functions due to the LNS platform, while other benefits such as the use of custom interfaces and the eradication of conversions to and from LNS arithmetic could be advantageous. Investigating the integration options for all rendering system components is of high priority in our future work.

Preliminary calculations show that overall speedups are reduced in comparison when the RTI acceleration hardware is used in a smart approach, which is due to the non-accelerated RAS traversal. The use of the RTI acceleration unit with RAS techniques, and RAS traversal acceleration, will also be looked at further in our future work.

8 CRITICAL ASSESSMENT

In this section we look at what has been achieved in this work. We begin by looking at the overall acceleration gained for photon mapping as a whole, before moving on to discuss the advantages and disadvantages of the two acceleration hardware devices designed and comparing these to existing offerings in the field.

There have been very few proposals for the acceleration of the complete photon mapping algorithm. It was found that, after analysis of the photon mapping algorithm, three core functions of the photon map renderer consume approximately 90% of total software execution time. These functions are those dealing with ray processing, RTI and RAS, and the photon search function used in the surface illumination calculations.

In this work we have designed two hardware devices to accelerate both the RTI and PS functions. The basis of both designs is high scalability, to the extent that multiple ICs each containing multiple of the RTI and PS designs can be used, and in the case of ray processing, the use of RAS structures can be eliminated. We have estimated that, using these devices, close to an order of magnitude speedup in overall photon mapping render time is achievable in comparison to our software implementation, which is the common implementation platform for the photon mapping algorithm. This figure does not include any adjustments for the differences in operating frequency of the two systems, which is 15 times lower for the RTI acceleration hardware and 9 times lower for the PS acceleration device compared to the software implementation executed on our 3.0GHz PC.

While the processing power of PCs will undoubtedly increase over time, due to the scalability of the acceleration hardware it is likely that speedups over equivalent software implementations can be maintained. Our calculations in section 7.3 show that in the unlikely case that processor speeds were increased to the extent that total software processing time decreased by a factor of 2, using the same number of hardware acceleration devices would result in only a small reduction in overall speedup. The scalability of the solution means that the original speedup figures can be reinstated with the addition of further acceleration hardware devices if required.

Our analysis of the photon mapping algorithm also showed that approximately 10% of total processing time, when executed in software, is spent in the ancillary main-line rendering functions. As a result of the reduction in time spent executing the core rendering functions when the acceleration hardware is used, which we have shown to be in the order of 42 times less, the proportion of time spent in these main-line tasks rises to approximately 80% of total execution time. For significant further speedups, these as yet non-optimised main-line rendering routines must be accelerated.

We will now look at the individual acceleration devices in more detail. Our primary efforts were concerned with the photon search hardware, due to the large proportion of time photon searching consumes during software renderings. There have been few proposals for the acceleration of the photon search function in hardware. Several of these approaches aim to rework the photon mapping algorithm into a form where indirect illumination can be calculated more quickly or easily, or in a completely different manner; while others look to accelerate a near traditional *k*D-tree search. Many of these solutions look to use GPUs, with only one design which directly accelerates *k*D-tree traversal having been prototyped on an FPGA device. All are limited in the scalability that they can offer.

Our aim was to provide a more scalable solution to the photon search problem. As photons are stored in memory, the obvious choice for a massively parallel processing architecture was CAM, where these photons would be processed locally. However, for this to happen, the processing hardware had to be minimal in size due the requirement for many parallel processing CAM cells. The processing of the photon locations was reworked in a novel way, where instead of looking for photons in a sphere; an AABB is used as the search volume. Simple magnitude comparisons are all that are needed to perform a photon-AABB inclusion test, which require only a fixed-point subtractor, which forms the basis on a LNS divider and a floating-point comparator.

The major advantage of our acceleration design is the scalability of the solution, due to the minimal hardware required to perform an AABB inclusion test. This means that for moderate hardware resources many photons can be tested for inclusion simultaneously. A possible issue with the system is that *k*NN searches cannot be performed in the hardware; however two possible solutions were identified and presented. If a *k*NN search is indeed required, the reduced result set returned from the CAM after a search can be refined; much as is done in a *k*D-tree based software search. This would be our preferred solution. Alternatively, separate global and caustic photon maps could be used, with different search bounds set for each.

We have shown that our CAM design could offer one to two order of magnitude speedups over a software photon search using statistics from IC synthesis of this device and the profiling results from a software rendering.

In comparison to other proposals for PS acceleration, we are unaware of any that make use of a CAM architecture, or indeed any highly parallel arrangement, to process AABB inclusion tests. In the context of RTI, there is one device that uses an intelligent memory architecture for the processing of ray intersections and can calculate illumination using radiosity [69, 70]. In terms of performance, little comparable data has been presented by existing proposals, which tend to compare results with a software based renderer much the same as the comparison made in this work.

With the costly photon search function accelerated, attention was turned to the optimisation of the still time consuming ray processing routines of RTI and RAS. Again, scalability was the driver behind the design, and for the sake of simplicity, we aimed to achieve sufficient acceleration using a RTI acceleration device such that RAS techniques would not be required. The vast majority of existing proposals make use of RAS techniques, and are limited in scale due to the use of large hardware structures or implementation using GPUs.

Many existing proposals also depend on ray coherence to be able to efficiently process packets of rays. Ray coherence is abundant in ray-tracing, however we have found coherency between rays in photon mapping to be much reduced. This is due to photons, which are specialised rays, often being scattered from light sources randomly, combined with the fact that there are usually many more secondary, and less coherent, rays to process due to the more complex ray-surface interactions employed in the most realistic of renderers.

After analysing various RTI methods to find the most efficient for implementation, it was found that all required a division operation which would be difficult and costly to implement in floating-point arithmetic. For this reason, a pipelined design was developed based on LNS arithmetic. This is a scalar unit designed to process ray-triangle intersection calculations, for a single ray, as quickly as possible. The processing efficiency of the design is not dependant on ray coherence, making it ideally suited to the processing of RTI calculations in photon mapping.

The advantages of our RTI hardware over existing offerings are in its scalability and also the non-dependence on ray coherence. This means that multiple RTI calculations can be performed per clock cycle by multiple RTI pipelines, and makes the acceleration unit ideal for use in a photon map renderer.

Our results show that up to an order of magnitude speedup over software ray processing time, which uses a *k*D-tree for a RAS, is achievable when using a brute-force approach. This level of speedup makes ray processing time in hardware similar to that of the photon search when executed in hardware. Therefore both devices can operate simultaneously, with total execution time being that of the device taking the longest. We have also shown that ray processing speedups can be maintained as model complexity increases, to a limit of say 100K to 150K triangles for an array of 64 or 32 ICs each containing 8 RTI pipelines.

Another advantage of the design is its versatility, making it adaptable to various RAS techniques, if such techniques are required. Only initial investigations have been undertaken, however preliminary estimates show that order of magnitude speedups in RTI processing time would be achievable if the RTI acceleration unit is used to directly replace the equivalent software function. In these calculations we assume that the same *k*D-trees as used during software renders are used as RAS, and that RAS traversal computation remains in software. To achieve this level of speedup for overall ray processing the non-optimised RAS traversal function would need to be accelerated.

A possible issue the RTI acceleration unit is the latency of the pipeline. However, speedups presented were actually for a worst case scenario where all pipelines must fully flush after the processing of a ray. One possible method to increase the utilization of the pipelines was described in section 6.1.3.5, and used multiple result registers indexed by a ray identifier.

In comparison to the majority of existing RTI acceleration offerings, our design mainly differs in scalability, but also differs in the arithmetic platform and RTI algorithm used. Many existing proposals for RTI acceleration devices are inherently limited in scale, as already discussed. A lot of these proposals are based on floating-point arithmetic and use a modified version of Arenbergs algorithm due to its efficiency when executed on this arithmetic platform. There is however one previous work detailing hardware which performs ray-object calculations in 20-bit logarithms [60], however few details on the intersection unit are provided.

In terms of performance, few results that detail the actual RTI computation rate are made available for most existing proposals, which tend to offer FPS rates instead. There is one recent proposal [71] where relevant performance data is provided however, where over several frames, 259,572 RTIs were performed by an FPGA based design in 31ms, equating to 119ns per RTI. Notably, as assumed in the calculation of our own results, no RAS technique was employed. In comparison, for a single RTI pipeline operating at 200MHz, a single RTI calculation can be performed in 5ns, which is over 20 times faster. If the number of pipelines is scaled to say 8, for example, a reduction in RTI processing time of almost 200 times would be possible.

In summary, two devices have been designed for acceleration of the RTI and PS functions of the photon mapping algorithm. Facilitated by the use of LNS arithmetic, both devices are highly scalable, in the case of the RTI unit to the extent that acceleration can be achieved over the software ray processing routines without the need for a RAS technique. The LNS arithmetic units used are proven to have a greater accuracy than their floating point equivalents, meaning that no degradation in image quality will result from the use of the acceleration devices. We have shown that a 42 times speedup over the software execution times of the core rendering functions is possible using these hardware devices, and that this advantage is likely to be maintained as processor speeds increase due to the scalability of the designs.

The reductions in core function execution time lead to close to an order of magnitude speedup in total photon map rendering time. For further reductions in overall execution time, the main-line rendering functions, which now account for a large proportion of this overall rendering time, should be targeted for optimisation.

9 FURTHER WORK

In this work we have presented hardware acceleration designs for two core functions of the photon mapping algorithm. Future work will be focused in the following areas:

Main-line Acceleration /System Integration

The major priority of our future work will be the acceleration of the main-line rendering functions. Our analysis showed that approximately 10% of total photon mapping execution time was spent performing these tasks during a software rendering. However, when our RTI and PS acceleration devices are used, the time spent in these main-line functions increases to approximately 80% of total execution time. For significant further acceleration of the photon mapping algorithm, it is these ancillary functions that should be targeted for optimisation.

One possible option for the acceleration of these main-line functions is that of using an LNS based processor for their execution, which would act as the rendering host. It has already been shown LNS arithmetic excels in areas of complex computation, making it a good candidate for use here. It is also unlikely that a fixed function hardware structure would be suitable for executing the main-line rendering functions, due to the large range of tasks that must be undertaken.

So far we have assumed that both acceleration hardware devices designed in this work will use high speed interfaces, such as IEEE 1394 or USB 3.0, to communicate with a host PC executing the main-line rendering tasks. If an LNS based processor was able to sufficiently accelerate the main-line rendering functions making it suitable as a rendering host, this would allow custom interfaces to be used between this and the two acceleration devices. As both acceleration hardware devices designed in this work are based on LNS arithmetic, this would remove the need for any number system conversions; however such conversions are rare due to the use of number system agnostic pointers to data in many cases.

Transmission of Search Results from PS CAM

In the PS CAM design presented, a single bit within a vector is used to represent each photon stored in a CAM cell. If any photon in a CAM cell is found to be within the AABB search volume, the bit associated with this photon is set. This search result vector is only read for a particular CAM cell if at least one photon is found to be within this cell during a search. Initially, a vector indicating whether each cell has at least one included photon is read from the CAM.

This hierarchical approach using a single bit per photon was taken to minimise the amount of data that must be read after each photon search, while still allowing included photons to be relocated in renderer memory. The alternative was to use a unique identifier per photon, which would be costly in terms of CAM storage and data transfer.

Our chosen method of search result storage and transfer is most efficient when many of the photons found during a search are stored within few CAM cells, meaning there are fewer search result vectors to transfer. In order to reduce the number of cells containing included photons, these photons should be allocated to cells based on spatial locality, as if one photon is included in an AABB the probability of spatially close photons also being within the AABB in high. Storing photons within cells in this manner has no impact on processing time when the fastest processing configuration is used, which is that of using 6 LNS dividers per CAM cell.

Further work will be undertaken relating to this spatially sensitive photon allocation with the aim of reducing search result transfer time.

RTI/RAS Integration and RAS Acceleration

The RTI acceleration hardware designed is versatile, and can operate both with and without a RAS technique, in smart or brute-force configurations. The aim of the RTI design was scalability, which would make sufficient acceleration of the ray processing functions possible without the use of RAS techniques. This was shown to be possible, however, in a brute-force approach several orders of magnitude more RTI calculations must be undertaken than a renderer where a RAS is employed. For this reason, large numbers of RTI pipelines are required.

This hardware requirement is significantly reduced if a RAS is employed. Preliminary investigations have shown that significant acceleration of the RTI processing function would be possible using as little as a single RTI pipeline. Further work will be undertaken on finding suitable RAS techniques for use with the RTI acceleration hardware designed, and determining the optimal methods of integrating these techniques for optimal performance. Acceleration of the RAS traversal functions required will also be investigated.

Summary of Further Work

In summary, a priority of our future work will be the acceleration of ancillary main-line rendering tasks of the photon mapping algorithm, which would lead to large increases in the overall acceleration achievable. This acceleration may be achieved as we integrate the two acceleration units into a complete photon map rendering solution, as it is possible that a custom dedicated rendering controller could be a good fit for executing the main-line rendering tasks. If this were to be based on LNS arithmetic, number system conversions between the various system components would be eliminated. Such a configuration would also have additional benefits in interfacing, allowing custom interfaces to be used between the various rendering system components.

Additional work will also be undertaken to improve the existing acceleration hardware designs. For the RTI acceleration unit, we will investigate RAS integration and optimisation which could lead to reductions in hardware requirement, while for the PS CAM design we will look to reduce the volume of data that must be transferred after a photon search.

10 CONCLUSIONS

Photon mapping is a computer graphics algorithm which excels in synthesising realistic images. The main difference between photon mapping and earlier ray based rendering methods is in the illumination calculation, in which a more accurate representation of the actual physics of light is used. This more realistic illumination calculation adds complexity to the rendering algorithm, making it one of the most costly in terms of execution time. The aim of this work was to reduce the time needed to produce realistic images using the photon mapping algorithm.

Our initial work was focussed on creating a software photon mapping rendering suite. This was composed of RTI acceleration structure generators and the rendering software itself. Like many photon mapping implementations, additional features to further enhance image realism were added. These included a Fresnel based reflection model and microfaceting. One further feature, a profiling routine, was added to give valuable information on processing time and number of executions of most of the functions within the rendering software.

This profiling tool was used as we analysed our photon mapping software. Various rendering and model parameters were varied, including model complexity, number of photons emitted from light sources, and also the photon search radius. Results from this analysis provided comparative data allowing the quantification of any acceleration achieved. While it is true that software timings may change with PC used for software execution, processing time based comparisons are common in such predominantly software based algorithms.

This analysis confirmed that, for all permutations of rendering and model parameters, three core functions of the photon mapping algorithm dominated execution time. These three functions are: ray-triangle intersection processing, the processing associated with RTI acceleration structure traversal, and photon searching, used in the calculation of illumination estimates. For significant acceleration of the photon mapping, it was these routines that would need to be optimised.

With the aim of this work now focussed on acceleration of these functions, many commonly used techniques employed for their execution in software were investigated. Existing proposals for hardware acceleration were also investigated.

Many of the existing proposals for photon search acceleration hardware were based on reworking the photon mapping algorithm so that the photon search could be performed more quickly, or easily. Often however, this had an impact of reducing the accuracy of the illumination calculation which could cause visible errors in rendered images, making them unsuitable for the synthesis of highly realistic images. The vast majority of these proposals made use of GPUs, making them inherently limited in scalability. To our knowledge there is only one existing custom hardware prototype. This uses custom tree traversal units for a *k*D-tree photon map [79]. This approach also lacks scalability due to the size of the traversal units.

For RTI processing, there have been many proposals, the vast majority of which are limited in scalability as they are based on the use of GPUs or large hardware structures. Many were found to require a pre-calculated buffer of rays which are ready to process, which is difficult in realistic renderers such as photon mapping as rays are often serially dependant. Most proposals were more applicable to ray tracing, which has many coherent, or similar, rays. This can be taken advantage of during processing, with the processing efficiency of many proposals dependant upon this characteristic. In photon mapping, this coherence is reduced due to the random scattering of photon rays and also the randomness exhibited in secondary rays due to techniques often employed more accurately model ray-surface interactions e.g. microfaceting.

In terms of RTI acceleration structure processing, it was found that many of the RTI proposals considered employed some form of RAS processing to reduce the number of RTIs to compute during a rendering. Some of these proposals use simple regular grids, while others use more complex *k*D-trees combined with traversal units. Notably, one proposal from Kim et al [71] was able to offer acceleration without the use of a RAS technique.

This work details acceleration designs for two of these functions, PS and RTI, both of which are based on scalability. This scalability is made possible through:

- The adoption of a brute force approach, allowing massive parallelisation of the required computation
- The adoption of LNS arithmetic elements for the processing requirement, which facilitate a reduction in processing hardware area and thus benefits the massively parallel approach
- The use of a novel CAM intelligent memory, combined with an alternative spatial point inclusion test, for photon searching

Notably, the scalability of both devices allows any advantage gained to be maintained in the presence of ever increasing processor speeds. Also of note is the fact that both acceleration devices make use of LNS arithmetic elements, which are intrinsically more accurate than their floating point equivalents. It follows that no degradation in image quality will result from the use of the acceleration devices.

As stated above, the photon search acceleration unit uses a novel massively parallel CAM architecture, and employs local processing within each of the CAM cells to determine whether each photon contained is within a search volume provided. CAM is the ideal architecture in this case, as photon locations must be stored in RAM anyway, and all of these photons must undergo the same test during a search.

The intention of the RTI acceleration design was that, due to the scalability of the solution, sufficient acceleration would be provided without any RAS technique. Due to the versatile configuration of the RTI design, the acceleration unit can however work both with and without a RAS technique, with elimination of the RAS build and traversal being an obvious advantage of the latter approach. The RTI acceleration hardware is a scalar unit, working with one ray at a time, performing all RTI calculations for this ray in parallel, and doing so as quickly as possible. This makes it ideal for use in photon mapping where rays are often serially dependant. The processing efficiency of the design is not dependant on ray coherence, unlike many existing proposals.

FPGA based prototypes have been created based on these designs allowing proof of concept testing to take place. These prototypes allowed several test images to be rendered, proving correct operation. VLSI devices have been designed in VHDL, and simulations and syntheses of these, together with subsequent reasoning about their application, have provided statistics allowing speedups over equivalent software functions to be calculated.

In operation, the PS acceleration hardware takes only a description of an AABB search volume as input. After completion of the search, a bit sequence allowing relocation of the found photons in host render memory is returned. We show that a one to two order of magnitude speedup is achievable in PS execution time over the equivalent function in a standard software renderer. The scalability of the system allows this improvement to be maintained, to a limit, over software counterparts in the presence of ever improving processor performance.

When operating in a brute-force mode, without any RAS technique, the RTI acceleration system we have described takes as input little more than the raw model data that would be output by any modelling utility, and can achieve significant acceleration, up to an order of magnitude, over the software ray processing functions. Again, it is likely that this advantage can be maintained as processor speeds increase.

Preliminary investigations show the number of RTI pipelines can be reduced if a RAS technique were to be used, while still providing a significant acceleration in RTI processing. Full details are yet to be worked out, however if a RAS were to be used, then RAS traversal processing would need to be optimised to gain any reasonable acceleration for total ray processing time.

Looking at the combined impact offered by the PS and RTI acceleration units, we have shown that when the devices operate simultaneously, a 42 times speedup is possible over the core rendering functions when executed in software. These calculations were based on the use of 32 RTI ICs each containing 8 and between 8 and 16 32K capacity PS CAM ICs, which could easily be assembled on a plug-in board or within a small discrete enclosure, and make no adjustments for the differences in operating frequency between the PC host executing the software and the hardware acceleration devices (up to 15 times difference).

These reductions in core function processing time lead to close to an order of magnitude speedup in overall rendering time, assuming the two acceleration devices are connected to a host PC running the main-line rendering tasks by multiple high speed interfaces. As already stated, due to the scalability of both acceleration devices, it is likely that this advantage can be maintained as the speeds of general purpose processors improve. In section 7.3 we show that in the unlikely case that processor speeds were increased to the extent that total software processing time decreased by a factor of 2, using the same number of hardware acceleration devices would result in only a small reduction in overall speedup. However, due to the scalability of the solution the original speedup figures can be reinstated with the addition of further acceleration hardware devices.

Due to the acceleration of the core rendering functions in hardware, the majority of overall rendering time is spent in the remaining main-line ancillary tasks in such a configuration. As already discussed, in our future work we will look to accelerate the main-line renderer functions in order to improve the acceleration in overall execution time. We will also look to manufacture the designed acceleration units and integrate these in to a complete photon map rendering system, where the use of custom interfaces could be advantageous. Other components of future work will include further investigation on the use of RAS techniques with our RTI acceleration unit due to the promise shown in reducing hardware requirement, and also minimising the volume of data returned after a photon search.

11 LIST OF REFERENCES

1. Jensen, H.W., 'Global Illumination using Photon Maps', Eurographics Workshop on Rendering Techniques, Porto, Portugal, 1996, pp. 21-30.
2. Jensen, H.W., 'Realistic Image Synthesis using Photon Mapping', A K Peters, Ltd., Natick, MS, USA, 2001.
3. Hall, D., 'The AR350: Today's Ray Trace Rendering Processor', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware - Hot 3D Presentations, Los Angeles, CA, USA, 2001.
4. Gouraud, H., 'Continuous Shading of Curved Surfaces', IEEE Transactions on Computers, Vol. 20, No. 6, 1971, pp. 623-629.
5. Phong, B.T., 'Illumination for Computer Generated Pictures', Communications of the ACM, Vol. 18, No. 6, 1975, pp. 311-317.
6. Luebke, D., Harris, M., Krüger, J., Purcell, T., Govindaraju, N., Buck, I., Woolley, C. and Lefohn, A., 'GPGPU: General Purpose Computation on Graphics Hardware', ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques - Courses, Los Angeles, CA, USA, 2004.
7. McKenna, M., 'Worst-Case Optimal Hidden-Surface Removal', ACM Transactions on Graphics, Vol. 6, No. 1, 1987, pp. 19-28.
8. Mulmuley, K., 'An Efficient Algorithm for Hidden Surface Removal', ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques, Boston, MA, USA, 1989, pp. 379-388.
9. Glassner, A.S., 'An Introduction to Ray Tracing', Morgan Kaufmann, San Francisco, CA, USA, 1989.
10. Cook, R.L., Porter, T. and Carpenter, L., 'Distributed Ray Tracing', ACM SIGGRAPH Computer Graphics, Vol. 18, No. 3, 1984, pp. 137-145.
11. Whitted, T., 'An Improved Illumination Model for Shaded Display', Communications of the ACM, Vol. 23, No. 6, 1980, pp. 343-349.
12. Kajiya, J.T., 'The Rendering Equation', ACM SIGGRAPH Computer Graphics, Vol. 20, No. 4, 1986, pp. 143-150.
13. Jensen, H.W., 'A Practical Guide to Global Illumination using Photon Maps', SIGGRAPH Courses, New Orleans, LA, USA, 2000.
14. Goral, C.M., Torrance, K.E., Greenberg, D.P. and Battaile, B., 'Modeling the Interaction of Light Between Diffuse Surfaces', ACM SIGGRAPH Computer Graphics, Vol. 18, No. 3, 1984, pp. 213-222.
15. Havran, V., 'Heuristic Ray Shooting Algorithms', Ph.D Thesis, Czech Technical University, Prague, Czech Republic, 2000.
16. MacDonald, D.J. and Booth, K.S., 'Heuristics for ray tracing using space subdivision', The Visual Computer: International Journal of Computer Graphics, Vol. 6, No. 3, 1990, pp. 153-166.
17. Wald, I., 'Realtime Ray Tracing and Interactive Global Illumination', Ph.D thesis, Saarland University, Saarbrücken, Germany, 2004.
18. Hunt, W., Mark, W.R. and Stoll, G., 'Fast kd-tree Construction with an Adaptive Error-Bounded Heuristic ', IEEE Symposium on Interactive Ray Tracing, Salt Lake City, UT, USA, 2006, pp. 81-88.
19. Popov, S., Georgiev, I., Dimov, R. and Slusallek, P., 'Object Partitioning Considered Harmful: Space Subdivision for BVHs', ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics, New Orleans, LA, USA, 2009, pp. 15-22.
20. Stich, M., Friedrich, H. and Dietrich, A., 'Spatial Splits in Bounding Volume Hierarchies', ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics 2009, New Orleans, LA, USA, 2009, pp. 7-13.
21. Möller, T. and Trumbore, B., 'Ray/Triangle Intersection Statistical Analysis' [Web Page], 1997, Available from: http://jgt.akpeters.com/papers/MollerTrumbore97/stats.html [Accessed 20th May 2010].
22. Arvo, J. and Kirk, D., 'A Survey of Ray Tracing Acceleration Techniques' *in* An Introduction to Ray Tracing, Morgan Kauffman, San Francisco, CA, USA, 1989, pp. 201-262.
23. Szirmay-Kalos, L., Havran, V., Balazs, B. and Szécsi, L., 'On the Efficiency of Ray-Shooting Acceleration Schemes', Spring conference on Computer graphics, Budmerice, Slovakia, 2002, pp. 97-106.
24. Born, M., Wolf, E., Bhatia, A.B., Clemmow, P.C., Gabor, D., Stokes, A.R., Taylor, A.M., Wayman, P.A. and Wilcock, W.L., 'Principles of Optics: Electromagnetic Theory of Propagation,

Interference and Diffraction of Light', Seventh Edition, Cambridge University Press, Cambridge, UK, 1999.
25. Cook, R.L. and Torrance, K.E., 'A Reflectance Model for Computer Graphics', ACM Transactions on Graphics, Vol. 1, No. 1, 1982, pp. 7-24.
26. Torrance, K.E. and Sparrow, E.M., 'Theory for Off-Specular Reflection From Roughened Surfaces', Journal of the Optical Society of America, Vol. 57, No. 9, 1967, pp. 1105-1114.
27. Haines, E., 'Point in Polygon Strategies' *in* Graphics Gems IV, Academic Press, San Diego, CA, USA, 1994.
28. Walden, P., 'Fastest Point in Polygon Test', Ray Tracing News, Vol. 5, No. 3, 1992.
29. Green, C., 'Simple, Fast Triangle Intersection', Ray Tracing News, Vol. 6, No. 1, 1993.
30. Spackman, J., 'Simple, Fast Triangle Intersection, part II', Ray Tracing News, Vol. 6, No. 2, 1993.
31. MacMartin, S., 'Fastest Point in Polygon Test', Ray Tracing News, Vol. 5, No. 3, 1992.
32. Shimrat, M., 'Algorithm 112: Position of Point Relative to Polygon', Communications of the ACM, Vol. 5, No. 8, 1962, p. 434.
33. Snyder, J.M. and Barr, A.H., 'Ray Tracing Complex Models Containing Surface Tessellations', International Conference on Computer Graphics and Interactive Techniques, Anaheim, CA, USA, 1987, pp. 119-128.
34. Badouel, D., 'An Efficient Ray-Polygon Intersection' *in* Graphics Gems, Academic Press, San Diego, CA, USA, 1990, pp. 390-393.
35. Möller, T. and Trumbore, B., 'Fast, Minimum Storage Ray-Triangle Intersection', Journal of Graphics Tools, Vol. 2, No. 1, 1997, pp. 21-28.
36. Amanatides, J. and Choi, K., 'Ray Tracing Triangular Meshes', Western Computer Graphics Symposium, Banff, Alberta, Canada, 1997, pp. 43-52.
37. Erickson, J., 'Plücker Coordinates', Ray Tracing News, Vol. 10, No. 3, 1997.
38. Jones, R., 'Intersecting a Ray and a Triangle with Plücker Cooridinates', Ray Tracing News, Vol. 13, No. 1, 2000.
39. Shoemake, K., 'Plücker Coordinate Tutorial', Ray Tracing News, Vol. 11, No. 1, 1998.
40. Arenberg, J., 'Ray/Triangle Intersection with Barycentric Coordinates', Ray Tracing News, Vol. 1, No. 11, 1988.
41. Schmittler, J., 'SaarCOR - A Hardware Architecture for Realtime Ray Tracing', Ph.D Thesis, Saarland University, Saarbrücken, Germany, 2006.
42. Glassner, A.S., 'Space Subdivision for Fast Ray Tracing', IEEE Computer Graphics and Applications, Vol. 4, No. 10, 1984, pp. 15-22.
43. Jansen, F.W., 'Data Structures for Ray Tracing', Eurographics Seminars on Data Structures for Raster Graphics, Steensel, The Netherlands, 1986, pp. 57-73.
44. Smits, B., 'Efficiency Issues for Ray Tracing', Journal of Graphics Tools, Vol. 3, No. 2, 1998, pp. 1-14.
45. Haines, E., 'Efficiency Improvements for Hierarchy Traversal in Ray Tracing' *in* Graphics Gems II, Academic Press, San Diego, CA, USA, 1991, pp. 267-272.
46. Goldsmith, J. and Salmon, J., 'Automatic Creation of Object Hierarchies for Ray Tracing', IEEE Computer Graphics and Applications, Vol. 7, No. 5, 1987, pp. 14-20.
47. Wald, I., Boulos, S. and Shirley, P., 'Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies', ACM Transactions on Graphics, Vol. 26, No. 1, 2007.
48. Fujimoto, A., Tanaka, T. and Iwata, K., 'ARTS: Accelerated Ray-Tracing System', IEEE Computer Graphics and Applications, Vol. 6, No. 4, 1986, pp. 16-26.
49. Amanatides, J. and Woo, A., 'A Fast Voxel Traversal Algorithm for Ray Tracing', Eurographics, Amsterdam, The Netherlands, 1987, pp. 3-10.
50. Lagae, A. and Dutre, P., 'Compact, Fast and Robust Grids for Ray Tracing', ACM SIGGRAPH International Conference on Computer Graphics and Interactive Techniques - Talks, Los Angeles, CA, USA, 2008.
51. Sung, K. and Shirley, P., 'Ray Tracing with the BSP Tree' *in* Graphics Gems III, Academic Press, San Diego, CA, USA, 1992, pp. 271-274.
52. Bentley, J.L., 'Multidimensional Binary Search Trees used for Associative Searching', Communications of the ACM, Vol. 18, No. 9, 1975, pp. 509-517.
53. Havran, V., Kopal, T., Bittner, J. and Žára, J., 'Fast Robust BSP Tree Traversal Algorithm for Ray Tracing', Journal of Graphics Tools, Vol. 2, No. 4, 1997, pp. 15-23.
54. Woop, S., Marmitt, G. and Slusallek, P., 'B-KD Trees for Hardware Accelerated Ray Tracing of Dynamic Scenes', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Vienna, Austria, 2006, pp. 67-77.

55. Gionis, A., Indyk, P. and Motwani, R., 'Similarity Search in High Dimensions via Hashing', International Conference on Very Large Data Bases, Edinburgh, Scotland, 1999, pp. 518-529.
56. Indyk, P., Motwani, R., Raghavan, P. and Vempala, S., 'Locality-Preserving Hashing in Multidimensional Spaces', ACM Symposium on Theory of Computing, El Paso, TX, USA, 1997, pp. 618-625.
57. Guttman, A., 'R-Trees: A Dynamic Index Structure for Spatial Searching', ACM SIGMOD International Conference on Management of Data, Boston, MA, USA, 1984, pp. 47-57.
58. Wald, I., Günther, J. and Slusallek, P., 'Balancing Considered Harmful—Faster Photon Mapping using the Voxel Volume Heuristic', EUROGRAPHICS, 2004, pp. 595-603.
59. Christensen, P.H., 'Faster Photon Map Global Illumination', Journal of Graphics Tools, Vol. 4, No. 3, 1999, pp. 1-10.
60. Wrigley, A., 'Method of and apparatus for constructing an image of a notional scene by a process of ray tracing', *Assigned to* Advanced Rendering Technology Limited, No. 5933146, *Issued by* United States Patent Office, 1999.
61. Hall, D., 'The AR250: A New Architecture for Ray Traced Rendering', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware - Hot Topics, Los Angeles, CA, USA, 1999.
62. ARTVPS, 'RayBox Overview' [Web Page], 2007, Available from: http://www.pixelution.co.uk/Products/ARTVPS_raybox.html [Accessed 07th August 2010].
63. Schmittler, J., Wald, I. and Slusallek, P., 'SaarCOR: A Hardware Architecture for Ray Tracing', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Saarbrucken, Germany, 2002, pp. 27-36.
64. Schmittler, J., Woop, S., Wagner, D., Paul, W.J. and Slusallek, P., 'Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Grenoble, France, 2004, pp. 95-106.
65. Woop, S., 'A Ray Tracing Hardware Architecture for Dynamic Scenes', Diploma Thesis, Saarland University, Saarbrücken, Germany, 2004.
66. Woop, S., Schmittler, J. and Slusallek, P., 'RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing', ACM Transactions on Graphics, Vol. 24, No. 3, 2005, pp. 434-444.
67. Woop, S., 'DRPU: A Programmable Hardware Architecture for Real-time Ray Tracing of Coherent Dynamic Scenes', Ph.D Thesis, Saarland University, Saarbrücken, Germany, 2006.
68. Woop, S., Brunvand, E. and Slusallek, P., 'Estimating Performance of a Ray-Tracing ASIC Design', IEEE Symposium on Interactive Ray Tracing, Salt Lake City, UT, USA, 2006, pp. 7-14.
69. Kobayashi, H., Suzuki, K., Sano, K., Kaeriyama, Y., Saida, Y. and Oba, N., '3DCGiRAM: An Intelligent Memory Architecture for Photo-Realistic Image Synthesis', IEEE International Conference on Computer Design, Austin, TX, USA, 2001, pp. 462-467.
70. Kobayashi, H., Suzuki, K., Sano, K. and Oba, N., 'Interactive Ray-Tracing on the 3DCGiRAM Architecture', ACM/IEEE International Symposium on Microarchitecture, Istanbul, Turkey, 2002, pp. 53-59.
71. Kim, S., Nam, S. and Lee, I., 'Fast Ray-Triangle Intersection Computation Using Reconfigurable Hardware', International conference on Computer vision, Rocquencourt, France, 2007, pp. 70-81.
72. Hanika, J. and Keller, A., 'Towards Hardware Ray Tracing using Fixed Point Arithmetic', IEEE Symposium on Interactive Ray Tracing, Ulm, Germany, 2007, pp. 119-128.
73. Carr, N.A., Hall, J.D. and Hart, J.C., 'The Ray Engine', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Saarbrücken, Germany, 2002, pp. 37-46.
74. Purcell, T.J., Buck, I., Mark, W.R. and Hanrahan, P., 'Ray Tracing on Programmable Graphics Hardware', ACM Transactions on Graphics, Vol. 21, No. 3, 2002, pp. 703-712.
75. Mahovsky, J.A., 'Ray Tracing with Reduced-Precision Bounding Volume Hierarchies', Ph.D Thesis, University of Calgary, Calgary, Canada, 2005.
76. Wächter, C. and Keller, A., 'Instant Ray Tracing: The Bounding Interval Hierarchy', Eurographics Symposium on Rendering, Nicosia, Cyprus, 2006, pp. 139-149.
77. Singh, S., 'The Photon Pipeline', International Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia, Kuala Lumpur, Malaysia, 2006, pp. 333-340.
78. Singh, S. and Faloutsos, P., 'The Photon Pipeline Revisited: A Hardware Architecture to Accelerate Photon Mapping', The Visual Computer: International Journal of Computer Graphics, Vol. 23, No. 7, 2007, pp. 479-492.

79. Heinzle, S., Guennebaud, G., Botsch, M. and Gross, M., 'A Hardware Processing Unit for Point Sets', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Sarajevo, Bosnia and Herzegovina, 2008, pp. 21-31.
80. Purcell, T.J., Donner, C., Cammarano, M., Jensen, H.W. and Hanrahan, P., 'Photon Mapping on Programmable Graphics Hardware', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, San Diego, CA, USA, 2003, pp. 41-50.
81. Czuczor, S., Szirmay-Kalos, S., Szécsi, L. and Neumann, L., 'Photon Map Gathering on the GPU', EUROGRAPHICS (Short Presentations), Dublin, Ireland, 2005, pp. 117-120.
82. He, X.D., Torrance, K.E., Sillion, F.X. and Greenberg, D.P., 'A Comprehensive Physical model for light reflection', ACM SIGGRAPH Computer Graphics, Vol. 25, No. 4, 1991, pp. 175-186.
83. Ma, V.C.H. and McCool, M.D., 'Low Latency Photon Mapping using Block Hashing', ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware, Saarbrücken, Germany, 2002, pp. 89-99.
84. McGuire, M. and Luebke, D., 'Hardware-Accelerated Global Illumination by Image Space Photon Mapping', ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics, New Orleans, LA, USA, 2009, pp. 77-89.
85. Coleman, J.N., Softley, C.I., Kadlec, J., Matousek, R., Tichy, M., Pohl, Z., Hermanek, A. and Benschop, N.F., 'The European Logarithmic Microprocessor', IEEE Transactions on Computers, Vol. 57, No. 4, 2008, pp. 532-546.
86. Haselman, M., Beauchamp, M., Wood, A., Hauck, S., Underwood, K. and Hemmert, K.S., 'A Comparison of Floating Point and Logarithmic Number Systems for FPGAs', IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, USA, 2005, pp. 181-190.
87. Lewis, D.M., 'An Accurate LNS Arithmetic Unit Using Interleaved Memory Function Interpolator', IEEE Symposium on Computer Arithmetic, Windsor, Canada, 1993, pp. 2-9.
88. Che-Ismail, R. and Coleman, J.N., 'ROM-less LNS', Accepted for publication at IEEE International Symposium on Computer Arithmetic, Tuebingen, Germany, 2011.
89. Arnold, M.G., Bailey, T.A., Cowles, J.R. and Winkel, M.D., 'Applying Features of IEEE 754 to Sign/Logarithm Arithmetic', IEEE Transactions on Computers, Vol. 41, No. 8, 1992, pp. 1040-1050.
90. Lewis, D.M., 'An Architecture for Addition and Subtraction of Long Word Length Numbers in the Logarithmic Number System', IEEE Transactions on Computers, Vol. 39, No. 11, 1990, pp. 1325-1336.
91. Lewis, D.M., 'Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit', IEEE Transactions on Computers, Vol. 43, No. 8, 1994, pp. 974-982.
92. Lewis, D.M., '114 MFLOPS Logarithmic Number System Arithmetic Unit for DSP Applications', IEEE Solid-State Circuits, Vol. 30, No. 12, 1995, pp. 1547-1553.
93. Swartzlander, E.E. and Alexopoulos, A.G., 'The Sign/Logarithm Number System', IEEE Transactions on Computers, Vol. 24, No. 12, 1975, pp. 1238-1242.
94. Taylor, F.J., Gill, R., Joseph, J. and Radke, J., 'A 20 Bit Logarithmic Number System Processor', IEEE Transactions on Computers, Vol. 37, No. 2, 1988, pp. 190-200.
95. Yu, L.K. and Lewis, D.M., 'A 30-b Integrated Logarithmic Number System Processor', IEEE Solid-State Circuits, Vol. 26, No. 10, 1991, pp. 1433-1440.
96. Coleman, J.N., 'Simplification of Table Structure in Logarithmic Arithmetic', Electronics Letters, Vol. 31, No. 22, 1995, pp. 1905-1906.
97. Coleman, J.N. and Chester, E.I., 'A 32-Bit Logarithmic Arithmetic Unit and its Performance Compared to Floating-Point', IEEE Symposium on Computer Arithmetic, Adelaide, Australia, 1999, pp. 142-151.
98. Coleman, J.N. and Kadlec, J., 'Extended Precision Logarithmic Arithmetic', IEEE Conference on Signals, Systems and Computers, Pacific Grove, CA , USA, 2000, pp. 124-129.
99. Coleman, J.N., Softley, C.I., Kadlec, J., Matousek, R., Licko, M., Pohl, Z. and Hermanek, A., 'The European Logarithmic Microprocessor - a QR RLS Application', IEEE Conference on Signals, Systems and Computers, Pacific Grove, CA , USA, 2001, pp. 155-159.
100. Coleman, J.N., Chester, E.I., Softley, C.I. and Kadlec, J., 'Arithmetic on the European Logarithmic Microprocessor', IEEE Transactions on Computers, Vol. 49, No. 7, 2000, pp. 702-715.
101. Coleman, J.N., Chester, E.I., Softley, C.I. and Kadlec, J., 'Corrections to Arithmetic on the European Logarithmic Microprocessor', IEEE Transactions on Computers, Vol. 49, No. 10, 2000, p. 1152.

102. Hoggins, C.A. and Coleman, J.N., 'Hardware Acceleration of Photon Mapping', ACM SIGGRAPH/EUROGRAPHICS Conference on High Performance Graphics (Poster Presentations), New Orleans, LA, USA, 2009.
103. Faraday Technology Corporation, 'Faraday Standard Cell Library FSD0A_A Data Book', Revision 0.2, Faraday Technology Corporation, Hsinchu City, Taiwan, 2006.
104. Faraday Technology Corporation, 'UMC/Faraday 90nm 2.5V with 3.3V tolerant generic I/O Data Book', Revision 1.0, Faraday Technology Corporation, Hsinchu City, Taiwan, 2006.
105. Faraday Technology Corporation, 'FXPLL110HD0A Phase-Locked Loop Data Sheet', Revision 0.6, Faraday Technology Corporation, Hsinchu City, Taiwan, 2007.
106. Xilinx Inc, 'DS099: Spartan-3 FPGA Family: Complete Data Sheet', Revision 2.1, Xilinx Inc, San Jose, CA, USA, 2006.
107. Enterpoint UK, 'RaggedStone1 User Manual', Revision 1.03, Enterpoint UK, Malvern, UK, 2006.
108. Enterpoint UK, 'Programming RaggedStone1 User Guide', Revision 1.0, Enterpoint UK, Malvern, UK, 2006.
109. Enterpoint UK, 'PCI I/O Expansion and Power Module', Revision 2.1, Enterpoint UK, Malvern, UK, 2006.
110. Cypress Semiconductor Corporation, 'CY7C1061AV33 - 1Mx16 Static RAM Datasheet', Cypress Semiconductor Corporation, San Jose, CA, USA, 2006.
111. Xilinx Inc, 'XAPP463: Application Note - Using Block RAM in Spartan-3 Generation FPGAs', Revision 2.0, Xilinx Inc, San Jose, CA, USA, 2005.
112. Faraday Technology Corporation, 'FSD0K_A|FSD0K_B Memaker Overview', Faraday Technology Corporation, Hsinchu City, Taiwan, 2008, pp. 1-8.