Integration of Ray-Tracing Methods into the Rasterisation Process

by

Shane Christopher, B.Sc. GMIT, B.Sc. DLIADT

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment of the

requirements for the Degree of

MSc. Computer Science

(Interactive Entertainment Technology)

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Shane Christopher

September 8, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Shane Christopher

September 8, 2010

Acknowledgments

I would like to thank my supervisor Michael Manzke as well as my course director John Dingliana for their help and guidance during this dissertation. I would also like to thank everyone who gave me support during the year and all my fellow members of the IET course for their friendship and the motivation they gave me.

Shane Christopher

University of Dublin, Trinity College
September 2010

Integration of Ray-Tracing Methods into the Rasterisation Process

Shane Christopher
University of Dublin, Trinity College, 2010

Supervisor: Michael Manzke

Visually realistic shadows in the field of computer games has been an area of constant research and development for many years. It is also considered one of the most costly in terms of performance when compared to other graphical processes. Most games today use shadow buffers which require rendering the scene multiple times for each light source. Even then developers are still faced with a wide range of shadow artefacts ranging from false shadowing to jagged shadow edges caused by low resolution shadow maps. In ray-tracing perfect, accurate shadows can be achieved but the performance cost involved has until now been considered too great to integrate into an interactive game. This dissertation will examine the best methods of using ray-tracing to calculate shadows and the consequences of doing so in terms of visual accuracy and resource usage. Improvements in hardware as well as research into scene management in dynamic ray-traced applications could make this a feasible alternative to current methods.

Contents

List of Figures

Chapter 1

Introduction

Ray-tracing and rasterisation, two distinct fields of computer graphics or just two different methods of doing the same thing? With the exponential increase in computing power it was inevitable that the areas of high quality and high performance rendering would start to merge into one. In recent years we have seen massive advances in the field of realtime ray-tracing however we are not yet at the stage where this can fully translate to the gaming industry.

The question that must be asked then is what benefits can be derived from these advances. Faster Graphical Processing Units (GPUs) have been the primary factor in the improvement of rasterised image quality to date. Much of this improvement has been achieved through vastly parallel programmable shader units accessible through languages such as the OpenGL Shading Language (GLSL) or directX's High Level Shader Language (HLSL).

These programmable, parallel processors can be used, not just for rasterisation shaders but for any task suited to a multi-threaded approach such as image processing where each pixel undergoes the same calculations. This method of processing is known as Single Instruction Multiple Data (SIMD). This processing architecture mixed with the power of current GPUs which at double precision can reach 928 gigaFLOPS [2] presents us with a perfect platform upon which to perform ray-tracing where there are millions of rays performing the same equations over and over.

1.1 Advantages

The obvious advantage to ray-tracing for any part of the graphics pipeline is increased image quality. Ray-tracing mimics the path of light through a scene to the extent of being refracted, reflected or even diffused into several rays. It does this in a very accurate way that just cannot be matched by current rasterised methods. For a good example of this let us look at a reflective sphere in a scene.

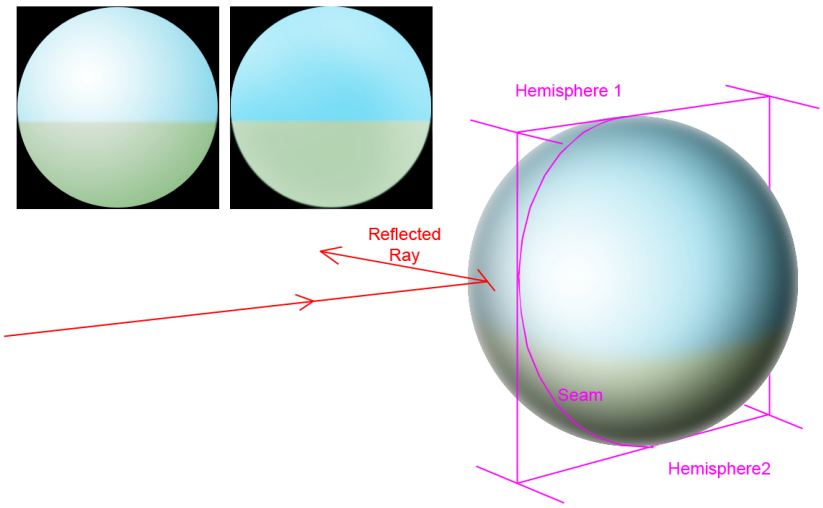

Figure 1.1: Reflective Sphere Example

In Figure 1.1 the view ray can be seen hitting the sphere and reflecting off at an angle determined by the surface normal. In ray-tracing all that needs to be done is continue the ray in its new direction to determine the reflective colour of the sphere at that point. This recursive method is simple and the exact same methods can be used for the primary and secondary rays such as the reflective ray in this instance. To avoid an infinite loop of reflections between two reflective objects it is possible to specify the maximum amount of bounces each ray can perform.

However in a rasterised renderer to create reflections it is necessary to render the entire scene into two or more textures. Common solutions are to use cube maps, six textures pointing up, down, left, right, forward and backwards from the object or

paraboloid maps which contain two 180° renderings of the scene facing in opposite directions which can be seen in the figure above. It is possible to read from these textures the correct reflective colour given the reflective ray direction however there are several visual artefacts produced.

Firstly when looking up the reflective texture only the direction is taken into account not the location on the reflective object. The reflected colour is what is seen from the objects centre point at that direction. This can produce highly inaccurate results when an object is very close to the reflective object but in the reflection it still looks too far away. The second problem with reflective maps is the resolution of the texture itself, if the texture is of low resolution then the resulting reflection will be low resolution too. If too high a resolution texture is used then memory issues will arise when there are several reflective objects in view. Another major issue occurs along the seams of the textures whether they are cube maps or paraboloid. There is often a visual difference where the different textures join together and multi-sampling also proves complicated and quite expensive to perform as several different textures must be sampled to get the neighbouring values.

This is just one example of where a ray-tracer vastly simplifies the rendering process and removes the need for secondary rendering of the entire scene. The same is true for refractive surfaces and shadow casting light sources. The latter, shadow casting will be the primary focus of this dissertation. Ray-tracing then seems to be the perfect rendering method however it does of course have many drawbacks of its own.

1.2 Disadvantages

The primary disadvantage of ray-tracing is quite simply one of performance. The time needed to test ray intersections against a mesh of triangles are many times greater then the time needed to rasterise the same mesh to the screen with current GPUs. There are just far more operations needed for the ray/triangle intersection test then there are to multiply the triangle vertices by the appropriate view and projection matrices used in rasterisation. Add to that the fact that since their introduction GPUs have concentrated solely on accelerating the specific operations needed in rasterisation so

that today much of the work of transforming triangles to fragments and on to pixels is hidden from shader programmers. These operations are hidden purely because they are an integral part of the GPU and as such have been optimised in hardware significantly.

If such time and research were dedicated to accelerating ray-tracing operations in hardware the differences in performance between the two would not be quite so great. However for now we must examine the possibility of performing ray-tracing on existing hardware. The code listing 1.1 is an example of a single ray-triangle intersection test in directX 9. One of the most expensive, in terms of computation time, parts of any shader is its texture samplers, as can be seen from the code for each triangle of a mesh four texture samples from a 128-bit texture are needed just to get the required information about the triangle.

```
1   float3 tri = tex2Dlod(ModelIndicesSampler, float4(linearToUV_Index(i), 0, 0)).xyz;
2   float3 v1 = tex2Dlod(ModelVerticesSampler, float4(linearToUV_Vertice(tri.x), 0, 0)).xyz;
3   float3 v2 = tex2Dlod(ModelVerticesSampler, float4(linearToUV_Vertice(tri.y), 0, 0)).xyz;
4   float3 v3 = tex2Dlod(ModelVerticesSampler, float4(linearToUV_Vertice(tri.z), 0, 0)).xyz;
5
6   float4 Nr = float4(normalize(cross(v3 - v1, v2 - v1)), 0);
7   Nr.w = dot(Nr.xyz, rayDir);
8   [branch] if (Nr.w < EPSILON)
9   {
10    Nr.w = dot(Nr.xyz, v1 - rayStart) / Nr.w;
11    [branch] if(Nr.w > EPSILON)
12    {
13      float3 p = rayStart + (rayDir * Nr.w);
14      float3 u = v2 - v1;
15      float3 v = v3 - v1;
16      float3 w = p - v1;
17      float4 UV_WV_VV_WU = float4(dot(u, v), dot(w, v), dot(v, v), dot(w, u));
18      float4 UU_den_s_t = float4(dot(u, u), 0, 0, 0);
19      UU_den_s_t.y = (UV_WV_VV_WU.x * UV_WV_VV_WU.x)-(UU_den_s_t.x * UV_WV_VV_WU.z);
20      UU_den_s_t.z = ((UV_WV_VV_WU.x * UV_WV_VV_WU.y)-(UV_WV_VV_WU.z * UV_WV_VV_WU.w))/UU_den_s_t.y;
21      UU_den_s_t.w = ((UV_WV_VV_WU.x * UV_WV_VV_WU.w)-(UU_den_s_t.x * UV_WV_VV_WU.y))/UU_den_s_t.y;
22      [branch] if (UU_den_s_t.z >= 0 && UU_den_s_t.w >= 0 && UU_den_s_t.z + UU_den_s_t.w <= 1)
23      {
24        return true;
25      }
26    }
27  }
```

Listing 1.1: Ray/Triangle Intersection in DirectX 9.0c

After these expensive texture lookups are performed it is then necessary to calculate the normal of the triangle then the dot product of the ray direction and the triangle normal. If this is less then zero then we know the normal is facing the origin of the ray. In this situation we are ignoring backfaces but with most game models it is assumed that back faces will be ignored so double faces are often used to ensure one face will

always be facing the camera. This is particularly useful in paper-thin areas such as leaves on a tree.

1.3 A Possible Solution?

For now it is not possible to use ray-tracing for every part of the rendering process although in the long-run it would be easier to do so. However that does not mean that ray-tracing can not still be used in some areas of the pipeline. This dissertation will examine the possibility of using ray-tracing to calculate shadows instead of using more traditional methods like shadow map buffers and stencil shadows. Using ray-tracing for shadows should give much crisper, more detailed shadows that do not contain the artefacts commonly found in other methods however it will be at the expense of performance. The ability to optimise this ray-traced shadowing method to tailor it for a game environment will also be looked at in detail. There may well be many compromises needed in order to come up with a solution that will run in realtime on todays hardware, and these compromises may produce their own artefacts, these will be studied and described and the pros and cons of each compromise shall be weighed against eachother.

Most rasterisation is still performed in one large pass today which calculates geometry, shading and lighting all in one go. It would be quite hard to fit ray-traced shadows into this as to shoot the secondary shadow rays we first need to know the starting position. In a full ray-traced solution this would involve shooting primary rays and finding their intersection points in the scene. However because we are combining ray-tracing with rasterisation we can use rasterisation to do what it does best, that is converting the triangles of the scene to the screen very quickly. If we render these triangles with the depth from the camera we can then rebuild the 3D position of each pixel on the screen to find the starting point of the secondary rays. However if a traditional forward renderer is used all the rasterisation information is lost. A separate pass would be needed to render the depth of each pixel, doubling the work of the rasteriser. We would also have to pass the shadow masks to the lighting system, adding to the amount of information already being passed through to this 'uber shader'.

Instead of a forward renderer we can instead use a deferred renderer. A deferred renderer splits the lighting pass from the geometry rasterisation. We can essentially hijack the G-buffer produced in between these passes and use the depth buffer inside of it to calculate our shadow rays and then pass the shadow masks to the lighting shader. Deferred shaders are gone into in more detail in section 3.2. The segmented style of a deferred renderer allows for much greater flexibility when it comes to inserting special effects. It is no longer needed to render a separate depth buffer for things like depth of field blurring or traditional shadow buffer mapping. It gives us an ideal basis from which to start and the complete encapsulation of the lighting pass lets us plug in either ray-traced shadows, buffer shadows or no shadows very quickly which helps vastly with benchmarking and examining the difference in performance between them.

1.4 The Goals

- To examine the potential of using ray-tracing as part of a modern game engine's rasterisation pipeline.
- To identify the most suitable scene acceleration methods for use in a highly dynamic game world.
- To experiment with optimisations intended to improve performance while maintaining image quality.

It should be clear that this dissertation is not aimed at creating perfect image quality nor is it expected that the end result will be suitable for games on todays hardware. We seek to look at what compromises can be made between image quality and performance, this means that the shadows created are not expected to rival those found in offline ray-tracing solutions or even other realtime ray-traced scenes involving a static scene or small amount of objects. Todays hardware is aimed purely at rasterisation, this is hopefully to change in the near future as developers start to integrate more and more ray-tracing elements into their game engines. It is therefore with caution that we must approach this issue, it is possible that the end result will be that it is just not feasible or recommended to attempt such a thing on todays hardware. However it is still desirable to examine the best ways of solving this problem even if the solution is

not applicable for several years.

In order to best solve this problem we must research past efforts in ray-tracing that have similar issues to a game environment. Large open areas, dynamic scenes and user interaction are just some of the issues that will seek to complicate or slow down the process. We must look for previous projects that have dealt with one or more of these issues and see how they solved these problems. Once done we must decide how best to take these solutions and combine them into one for our own needs. It is not expected that the end result will be perfect by any means, the time is not available to research fully every issue or spend the time tweaking and optimising every part of the project but even if just a small start can be made here on the issue of ray-tracing in a game world we can rest assured others in the coming years will certainly further this interesting and some would say necessary field of research.

Chapter 2

Background and State of the Art

2.1 Background

Currently the vast majority of real-time games on the market today use rasterisation only as their rendering methods. Rasterisation is the process of turning data structures representing the objects in the scene, usually as a list of triangles into pixels on the screen. This area has been heavily used and researched in the last 20 years or so which has lead to very fast, efficient and, in some cases, beautiful rendering of the game scene. Some of the most visually advanced solutions currently on the market can be seen in Crytek's CryEngine 2 [3] and id Software's Unreal Engine 3 [4].

Techniques such as high dynamic range rendering, screen-space ambient occlusion and advanced physics and animation engines produce relatively realistic results in many cases. However there are drawbacks to the rasterisation process, primarily the inaccuracies inherent in the approximation of lighting specifically in reflection and refraction methods. The most common method of approximating reflection is to use a cube map [5] which is a rendering of the surrounding scene along the positive and negatives of the three main axes. The reflection angles are then retrieved when drawing each pixel and the light of the cube map is retrieved. This works reasonably well in many scenes but has many artefacts around the edge of the cube map, at extreme angles and most noticeably in the lack of scale in the reflection map.

To combat these artefacts a different rendering method must be used. Ray-tracing is an ideal solution as it accurately replicates the path of light through the scene. Both reflection and refraction are inherently achieved by this method. Ray-tracing as a rendering technique is even older then rasterisation being first introduced by Arthur Appel in 1968 as ray-casting [6]. However even with the massive increases in hardware speed in recent years the best real-time ray-traced applications [7] are limited in scope and resolution and are still too slow for use in the game industry as a whole.

Several steps are needed in order to reduce the computation time of the rendering process. This dissertation plans to vastly reduce the amount of rays shot by using rasterisation for most objects in the scene and ray-tracing only for secondary shadow rays. We must also research the most efficient methods of ray traversal [8] and the various scene hierarchy solutions in use such as bounding volume hierarchy [9] or interactive k-D trees [10]. Luckily there are a wealth of papers and resources available that have examined these areas in minute detail. One paper that stands out as being particularly useful is the thesis of Martin Christen from the University of Applied Sciences Basel [11].

However, while there has been some talk on combining ray-tracing and rasterisation [12] [13] there are few papers detailing the integration of the two methods. The few that do examine this generally use non-consumer hardware or use scenes that are not equivalent to those found in games. One such paper by Ingo Wald and Philipp Slusallek [14] comes close to the method proposed in this dissertation. This paper is from 2001 yet there has still not been much research into the hybrid rendering process. This dissertation hopes to show whether or not a hybrid renderer is feasible in current game architecture and environments on consumer hardware.

2.2 Scene Acceleration Structures

The most basic form of ray-tracing can be accomplished by testing the ray against every triangle in a scene. This brute-force approach would work but is extremely slow. When we consider that most game scenes can consist of 10 million triangles or more and we would be shooting over 1 million rays this quickly mounts to a ridiculous 10

trillion intersection tests per frame.

In order to tackle this problem the scene can be split up so that the ray only performs intersection tests on a small part of the scene at a time. If an intersection is not found in that section of the scene then the ray moves on to the next section that it passes through. The different ways that attempt to achieve this are known as scene acceleration methods. We will be examining a few of the more popular methods as well as looking at their suitability when used in a realtime game environment.

Broadly speaking the different scene acceleration methods can be split up into object based and location based categories. Object based acceleration methods rely on the relativity of objects to each other. The most widely used example of this is bounding volume hierarchies[15] which we will look into in greater detail. Location based methods use the location of objects in a scene to determine whether or not to test them. The majority of the more popular location based solutions use some type of grid layout which the ray traverses through testing each grid node until an intersection is reached.

Both of these types of methods have their own advantages and disadvantages. One of the main differences between the two is often described as the "teapot in a stadium" problem. This problem is detailed very well by Eric Haines [16] who describes the use of an octree in such a situation as "a total failure". While perhaps not as dire as described in his article the octree would indeed perform much worse than a bounding volume hierarchy in this instance. In any form of grid the minimum grid size is the most important issue in such situations. Any ray that passes through the teapots grid node would have to test intersections against it, the larger the minimum grid size is the more rays there are that will have to test against it. In a bounding volume hierarchy however the scene would be separated into individual objects, the stadium, most likely would be separated into smaller pieces and the teapot would be on its own. Only the rays intersecting with the objects bounding volume would then proceed to test intersections against the objects mesh. This would result in a much faster solution in this situation then a grid-based one.

2.2.1 Bounding Volume Hierarchies

A bounding volume hierarchy (BVH) as described briefly above is an object based scene description. The hierarchy, at its lowest level is comprised of individual objects and their bounding volumes. As we move up through the levels nearby objects are grouped together and their bounding volumes merged into one. This continues until the groups are spread too far apart for a merged volume to produce a better result. During intersection testing the ray is first tested against the highest level of the hierarchy, determining which of the largest volumes it intersects with, after this the ray tests against the next level down in those volumes. This continues until the lowest level of the mesh with the individual objects, if the ray tests true against any of these a mesh intersection test is performed starting with the nearest objects.

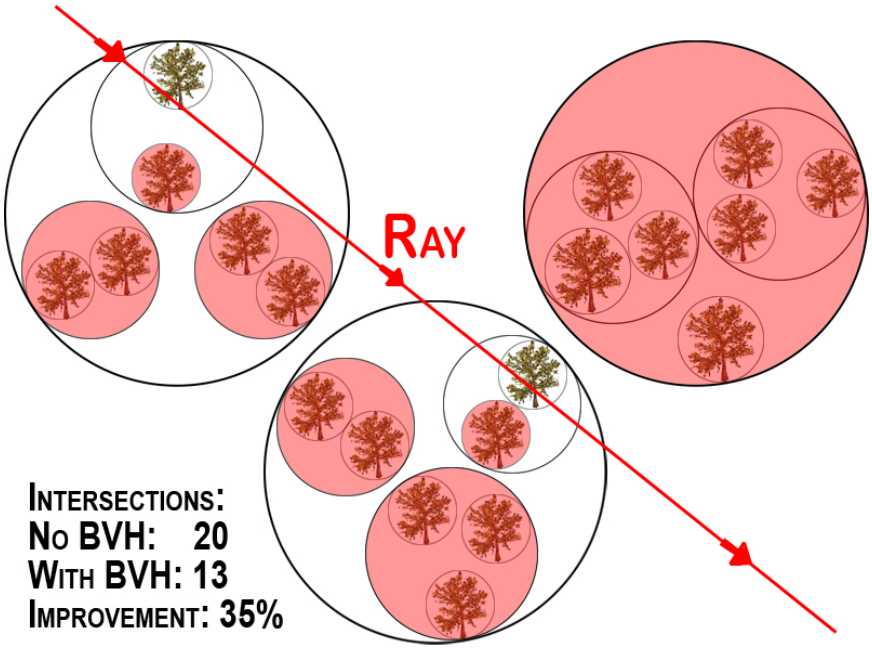

Figure 2.1: Example of a Bounding Volume Hierarchy

In figure 2.1 we can see that there are 3 layers of hierarchy. The highest layer being comprised of 3 bounding spheres. The volumes that are discarded are highlighted in red and at the end of testing against the hierarchy just two objects remain to be tested against. There are 20 objects in the scene and without a BVH the ray would have had to perform tests against the bounding volumes of all these, however

with the BVH only 13 tests are needed, this is an improvement of 35% for this example.

We can see from this that BVHs are perfect for clusters of objects spread across a large scene, this sounds perfect for a game environment and it would be if not for the cost in constructing the hierarchy. We can see from the results in Walds study of BVH construction methods [17] that the fastest achievable hierarchy construction time for the scenes tested was 36ms. This was for quite a simple scene of 174K triangles. For a dynamic game scene with a lot of fast moving objects the hierarchy would have to be rebuilt every frame to have stable results. To spend even 36ms a frame on rebuilding the hierarchy would cripple any game engine and make it unplayable.

There may well be research in the future that will help speed up hierarchy construction for dynamic scenes. Many of the current construction methods are based around grouping individual triangles in the scene, this is not suitable for a dynamic environment so perhaps object based BVH construction methods will prove to be faster where just the objects themselves need to be grouped together. This is an area that deserves more research, it might well prove to be perfectly feasible to construct a BVH using just object bounding spheres and not individual triangles. However the scope of this project does not allow for enough time to be given over to this matter.

In the end, when looking for a solution that will translate quickly and easily to a texture a bounding volume hierarchy just cannot match a grid structure. A grid by its very nature is similar to a texture and so translation between the two is much more intuitive. So we shall move on from bounding volume hierarchies and examine some of the grid based acceleration structures currently popular in realtime ray-tracing.

2.2.2 k-d Trees

A k-d tree is a "space-partitioning data structure" [1], it is similar in nature to an octree. The form of a k-d tree can be seen in figure 2.2. Each node, starting with a node encompassing the entire scene is split into two parts at the median location of all the objects in that node. This is continued until a limit is reached or until each node has just one object inside of it. In the figure below the first split can be seen in red,

the second in green and the third in blue. Note that unlike a quad tree or octree the nodes are split into just two parts, not four.

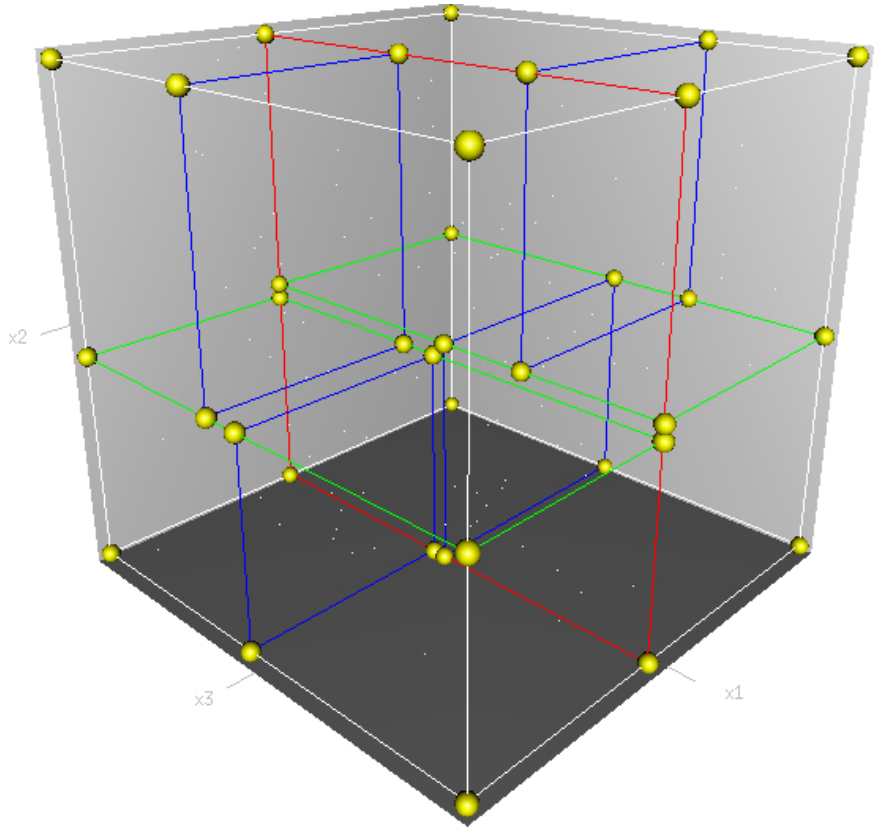

Figure 2.2: Example of a k-d tree. Used under GNU GPL license from wikipedia[1].

Such a method, unlike a standard grid, ensures that objects are evenly distributed throughout each of the child nodes. This removes the necessity of traversing the ray through potentially empty nodes for great distances. Any large empty area would be comprised of just one node thus vastly reducing the cost of traversal. However, like bounding volume hierarchies we must examine the cost of construction for a dynamic scene. Many realtime ray-tracing demonstrations use a purely static scene which negates the need to re-construct the acceleration structure each frame, these cannot then be considered relative to a game world where many objects will be moving and switching nodes each frame.

Unfortunately the construction of the kd-tree is also very costly in the amount of time taken to build it. Even with a highly optimised solution the time taken to build a scene of 1 million triangles can take around half a second[18] and this is with significant loss of performance during the rendering phase. We can see straight away that such a method is not suitable for a dynamic interactive world. There is also the issue of how well would a kd-tree translate to a texture in order to be passed to the GPU. A kd-tree is essentially a binary space partition, that is each node is repeatedly split in two. It should be quite feasible to copy such a structure into a texture however extra data would have to be copied for each node split.

In order to traverse through the nodes the location of each split in each node must be known, this would have to be attached to the texture somehow or placed in an additional texture. The matter of traversal itself is not as straight forward as other grid based methods. A good description of a realtime traversal method can be found in the paper 'Interactive k-D Tree GPU Ray-tracing'[10] which is also based around directX 9 code running on the GPU. These factors all combine together to make a kd-tree acceleration structure presently unfeasible for an interactive game environment.

2.2.3 Uniform Grid

A uniform grid is one of the simplest of acceleration structures in terms of both its construction, its traversal and its general structure. It is simply a grid of nodes spread uniformly across the scene each containing a list of objects within it. This grid structure can either be 2D or 3D depending on the environment it is used in. Objects are placed into the grid when they are created, the code behind checking which node the object should be in is quite simple, at its most basic form it is just a division of the objects position by the amount of nodes on each side of the grid. Each frame the object checks if the node it is in has changed, if it hasn't then nothing more needs to be done. If however the node the object is in has changed its reference must be removed from the previous node and placed into the new node. Both of these operations are generally quite quick but this depends on the amount of objects per node as removing an object reference necessitates the shifting of all object references after it in the array back one so that no null references appear before they should.

The null reference is used when intersecting against the nodes object list. The ray cycles through each object reference until a null or 0 reference is found, it is then considered that all objects in the node have been tested and the ray can move on to the next node. The DDA traversal method is gone into in more detail in the next section. This type of grid, because of its uniformity is perfect for transferring to a texture. All that is needed is a texture the size of the amount of nodes on a side multiplied by the square root of the amount of objects in a node. Each node in the texture looks like a square block of pixels where each pixel is an object reference id.

Due to the very fast construction and update speeds as well as the simplicity of conversion to a texture and the quick DDA traversal algorithm used by this grid it is very well suited to use in our dynamic scene. There are however some downsides to using a uniform grid. The biggest one is, as mentioned earlier in this chapter the "teapot in a stadium" issue. Once a ray enters a node it must check every object in that node until an intersection is hit or there are no more objects left to test. This can be optimised by doing a pre-test against the objects bounding sphere but even so it can still be very slow. There is also the matter of wide open areas still taking time during traversal. Even though there are no objects in these areas the traversal algorithm must still check using texture lookups and dynamic branching whether there is an object in the node or not.

Another major downside of a single-layer uniform grid is the necessity to use multiple object references in all the nodes a large object covers. Even smaller objects simply laying across a corner point between four nodes requires a reference to be placed in each of those nodes. This can lead to multiple intersection tests on the same object which in some situations can cripple a ray-tracers performance.

2.2.4 Customised Hierarchical Grid

In order to avoid the multiple intersection test problems presented by a single layer uniform grid a hierarchical structure can be used. The hierarchy is constructed of several layers of grids where each grids nodes increase in size in a similar way to a

quad-tree. Objects are first tested to see if they will fit into the lowest layer of the hierarchy where the node sizes are smallest. If they cannot fit into a node that size or if their bounding sphere overlaps into a neighbouring node then they are bumped up to the next level of the hierarchy and the tests repeated for the new larger node sizes. Eventually a massive object or an object overlapping at the middle of the grid will be bumped up as far as the highest hierarchy layer where the node size covers the entire scene.

In each hierarchy layer the amount of objects capable of being stored in each node is kept the same. This can be changed if there are a lot of huge objects in the scene but in most games there won't be that many of such a size, even a large building or skyscraper should fit into one of the lower layers. This type of structure is more commonly found in realtime physics solutions where it is known as "Hierarchical Spatial Hashing"[19][20] but is equally applicable in the field of ray-tracing. A lot of research areas like this are shared between the two fields of study and often a breakthrough in ray-tracing can help physics engines and vice versa.

When testing the shadow rays against the hierarchy we start first at the highest layer where the largest objects are held, we use exactly the same shader to perform the intersection tests and traversal for each layer which simplifies things greatly. The only differences when computing are the required values for the size of the grid and the nodes within and of course the grid texture itself. Those pixels which are shadowed by the higher layers of the hierarchy then do not need to be tested in the lower layers. For example in an urban scene if a large building is completely overshadowing a street below there is no need to test the pixels in its shadow against smaller objects as we know they are already occluded from the light. This should greatly improve performance especially in urban scenes with great blocky buildings that are easy to test against.

The hierarchical grid used in this test is a 2D grid, this will improve traversal speeds and also simplify the placement of objects within it. The decision was made to use a 2D grid because of the fact that in a game environment most objects that will cast shadows are usually in close proximity on the vertical axis. There is little point to having an entire vertical set of grid nodes when the vast majority of the objects in the

scene will be stored in just one or two of them. Having just a 2D grid does lead to the problem of a very high ray still testing against all the objects it passes over. To alleviate this each node has its maximum height recorded, this is the maximum height at which a ray passing through the node could intersect with any of the objects within it. This is calculated per frame as new objects are added to the node, it is simply the objects vertical position plus the radius of its bounding sphere. If the rays entry and exit points in the node are both above the maximum height then the node can be discarded and traversal can continue on to the next node. There is also an overall maximum height for the entire scene, once a ray goes above that point it is discarded as it will never intersect with an object after that. This is particularly useful when calculating shadows for sunlight as when the sun is overhead rays quickly reach the maximum height but would keep going and testing against nodes they traverse through if not for this check.

With these optimisations using a hierarchical grid has proved to be a very efficient scene acceleration structure to use in our game like environment. While it is not ideal in some situations as described above the efficiency of the traversal algorithm used somewhat negates that downside.

2.3 DDA Traversal

A differential analyser is an analogue computer designed to solve differential equations by integration[21]. It is an example of one of the earliest forms of computers with its historic roots as early as 1836. Differential analysers started to see widespread use in the early 1900s in a range of fields such as in fire control systems for naval gunnery[22]. Analogue differential analysers started to give way to digital ones with the rise of the transistor and microchip in the 50s and 60s.

Digital Differential Analysers (DDAs)[23] perform precisely the same task just in a digital programmable way. In the field of computer graphics DDA algorithms are most widely used as a method to draw lines and for linear interpolation of values across a range[24]. When traversing a uniform grid the same algorithm can be used to find every node which the ray passes through. In figure 2.3 we can see how this is accomplished.

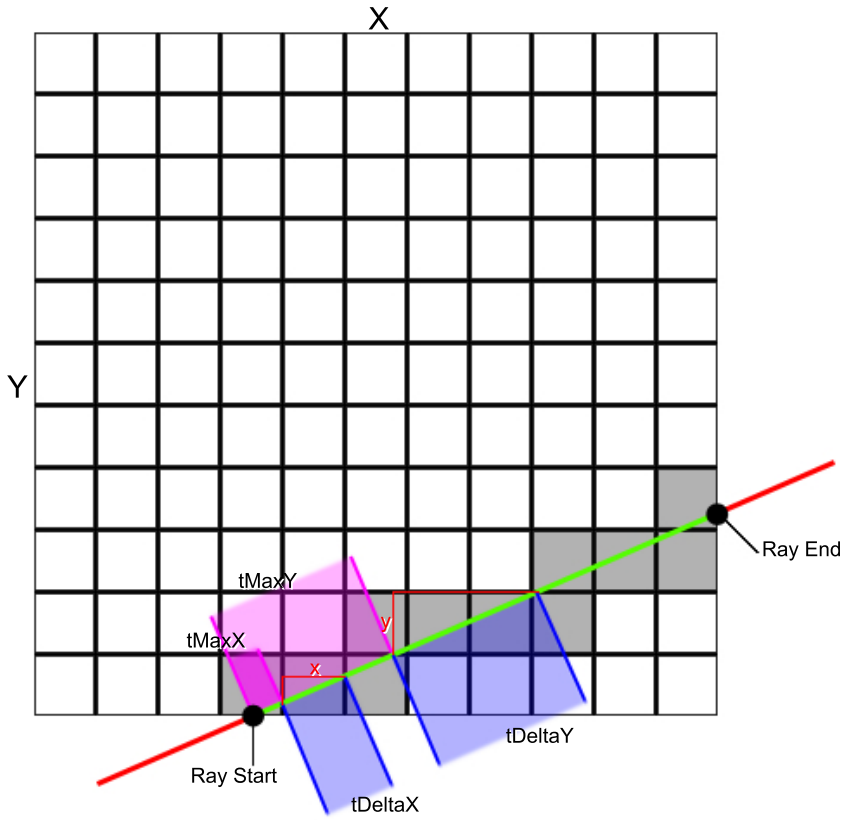

Figure 2.3: DDA Traversal Method.

Four values need to be determined when the algorithm is begun, the first two are tDeltaX and tDeltaY. These do not change during traversal, they are the distance the ray travels along its length while crossing one unit of the X and Y axes. In listing 2.3 these two values can be seen marked in blue. The distance along the ray is usually measured in time, or t for short, so if we imagine a photon of light traveling down this ray, tDeltaX is the value of time it takes for it to travel between 0 and 1 on the X axis.

The next two values needed are tMaxX and tMaxY, these change during each step of the traversal but at startup they are the distance from the ray start to the nearest intersection along the ray with each of the respective axes. These can be seen in the figure marked in magenta. Additionally we need to determine the direction of stepping, this is based on the direction of the ray. If the ray directions X component is positive

then stepX will be 1, if it is negative then stepX will be -1.

The algorithm is quite simple, it finds the minimum of the two values tMaxX and tMaxY, if tMaxX is the minimum value then tDeltaX is added to it and the rays current node is stepped forward by the amount on stepX. Alternatively if tMaxY is minimum then the same is done with the Y values. The new node can then be checked for an intersection, if there is none then we simply loop through the algorithm again, stepping the ray forward one node at a time until the end is reached or an intersection is found. The HLSL code used for DDA traversal in our implementation is included below.

```
1  //DDA VARIABLES, T AND T2 HOLD THE DISTANCES WHERE THE RAY ENTERS AND EXITS THE CURRENT NODE
2  float stepX, stepY, tMaxX, tMaxY, tDeltaX, tDeltaY, t, t2;
3  if(rayDir.x > 0.0f)
4  {
5    stepX = 1.0f;
6    tMaxX = (currNode.x + 1 - (rayStart.x / nodeSize)) / rayDir.x;
7    tDeltaX = 1.0f / rayDir.x;
8  }
9  else
10 {
11   if(rayDir.x < 0.0f)
12   {
13     stepX = -1.0f;
14     tMaxX = ((rayStart.x / nodeSize) - currNode.x) / rayDir.x;
15     tDeltaX = 1.0f / rayDir.x;
16   }
17   else
18   {
19     stepX = 0.0f;
20     tMaxX = 0.0f;
21     tDeltaX = 0.0f;
22   }
23 }
24 if(rayDir.z > 0.0f)
25 {
26   stepY = 1.0f;
27   tMaxY = (currNode.y + 1 - (rayStart.z / nodeSize)) / rayDir.z;
28   tDeltaY = 1.0f / rayDir.z;
29 }
30 else
31 {
32   if(rayDir.z < 0.0f)
33   {
34     stepY = -1.0f;
35     tMaxY = ((rayStart.z / nodeSize) - currNode.y) / rayDir.z;
36     tDeltaY = 1.0f / rayDir.z;
37   }
38   else
39   {
40     stepY = 0.0f;
41     tMaxY = 0.0f;
42     tDeltaY = 0.0f;
43   }
44 }
45 //ENSURE THE DDA VARIABLES ARE ALL POSITIVE
46 tMaxX = abs(tMaxX);
47 tMaxY = abs(tMaxY);
```

```
48  tDeltaX = abs(tDeltaX);
49  tDeltaY = abs(tDeltaY);
50  float y = rayStart.y;
51  //DDA LOOP EXITS WHEN THE RAY GOES OUTSIDE THE MAP OR FINDS AN INTERSECTION
52  [loop] while(currNode.x >= 0 && currNode.y >= 0 && currNode.x < nodeAmount && currNode.y <
        nodeAmount && intersected == false && y < maxHeight && t < MAX_SHADOW_DIST)
53  {
54    //DDA LOOP
55    if(tMaxX < tMaxY)
56    {
57      tMaxX += tDeltaX;
58      t += tDeltaX;
59      currNode.x += stepX;
60    }
61    else
62    {
63      tMaxY += tDeltaY;
64      t += tDeltaY;
65      currNode.y += stepY;
66    }
67    //FIND THE POINT WHERE THE RAY EXITS THE NODE
68    if(tMaxX < tMaxY)
69    {
70      t2 = t + tDeltaX;
71    }
72    else
73    {
74      t2 = t + tDeltaY;
75    }
76    y = min(rayStart.y + rayDir.y * t, rayStart.y + rayDir.y * t2);
77    //CHECK NODE INTERSECTION
78    intersected = checkNode(currNode, rayStart, rayDir, y);
79  }
```

Listing 2.1: DDA traversal in DirectX 9.0c

2.4 Realtime Ray-Tracing

There has been a lot of research into ray-tracing since Appel's very first paper in 1968. In the last decade or so the idea of realtime ray-tracing has become more and more popular in academic circles. Much of this research however is aimed towards purely scientific testing of one method against another and not the application of these methods in an interactive, dynamic environment. There is one project in particular that does take on this challenge however. The 'Enemy Territory: Quake Wars'[25] conversion to realtime ray-tracing by Intel's research team is attempting to prove that ray-tracing is an alternative in game rendering. Their ray-tracing is processed on very powerful multi-core CPUs available to them as Intel is of course the current leader in CPU development today.

Unfortunately there are not a lot of details on how their ray-tracing is performed or if

rasterisation is used for any portion of the rendering process. It would appear from the few articles that they have available that their solution is purely ray-traced. Much of the information that is available on the project is unfortunately highly biased towards ray-tracing and avoiding rasterisation at all costs. This is predictable when performed by a company dedicated primarily to developing CPUs and integrated GPUs.

2.5 Other Hybrid Solutions

As has been mentioned before there hasn't been a lot of academic research into the combination of rasterisation and ray-tracing in a game environment, however if we look further afield we can find examples of such hybrid combinations in other places. One example of a hybrid rendering solution is Pixar's solution to rendering many of their recent films which combines REYES[26] their scanline rasterisation algorithm with ray-tracing for some particular parts of the scene. One example of a film they used this approach in is 'Cars'[27]. For this movie they needed accurate reflections, refractions and shadows.

Previous methods of accomplishing this with a purely REYES based rendering system would have used environment maps just like those seen in realtime games today. However in order to calculate more complex and realistic visual details such as inter-reflectance where parts of a model reflect off eachother ray-tracing was needed. Rendering the entire film in ray-tracing however was not desired, not only would it be much slower but it would mean leaving behind the REYES renderer which they had worked on, optimised and perfected since the early 80's. The solution was to combine the two into a hybrid renderer. The REYES algorithm would operate first, rendering the image to a buffer, after this the objects marked for needing ray-tracing would have this ray-tracing calculated for them, the primary rays need not be shot again as the REYES rasterisation had already worked these out. This is very similar to the method used in our own solution.

There have been one or two theses on hybrid rendering before, one of these which stands out for its similarities is a thesis by Henrik Poulsen[28]. In his thesis however he approaches the problem merely as a proof of concept for the combination of the two

rendering methods. He uses a very simple, small scene with no acceleration structure and little other optimisations. He also calculates just shadow rays on the GPU but he uses a small amount of multi-sampling to produce noisy soft shadows.

Such a setup is not suitable for comparison to an interactive, dynamic game world however. The scene is far too small and simple for the performance values to be comparable. Even so the framerates are surprisingly similar to our own results when the amount of rays shot is taken into account. The only optimisation made by poulsen is a bounding sphere check before performing the mesh test. His results for roughly the same amount of shot rays as in our solution is around 30fps, close to our own average. This just serves to show the benefits of the fast traversal algorithm used in our application.

Chapter 3

Design and Implementation

3.1 Design

Most rasterisation engines today operate as forward-renderers, that is they calculate vertex positions, pixel positions, texturing and lighting all in one pass. In recent years, especially with the release of STALKER[29] a new method called deferred rendering has become more popular. Deferred rendering essentially performs all the geometry stages of the pipeline but defers the lighting stage to a secondary pass after each object has been drawn to the screen. The main reason for doing this is to avoid expensive light calculations for fragments that are occluded by objects in front of them.

If we take an example of two spheres, one small one behind one large one. In a forward renderer the small one would render itself to the screen, calculating all its vertex positions then performing all the pixel operations including lighting. The larger sphere would then draw itself to the screen and completely overdraw the sphere behind it. What has happened here is that a lot of expensive calculations have been performed for no benefit to the final image. In that same situation in a deferred renderer the small sphere would draw only its colour, depth, specular information and normals to the buffer. These do not take much time to calculate when compared to lighting calculations. The larger sphere would then draw over the small sphere and we would be left with a buffer containing only the larger spheres information. This information would then be used to perform lighting calculations only on the pixels still present in

the buffer.

This method of rendering is far more efficient in scenes where a lot of overdraw exists or scenes with many light sources. Further optimisations can be made such as drawing a sphere to represent a point light and only performing light calculations on pixels underneath that sphere. Deferred rendering also has benefits for what we wish to achieve in this test. The pre-existing g-buffer with each pixels depth provides all the information we need to calculate that pixels 3D position in the world, that is the shadow ray starting position and the direction to the light source is used as the direction of the shadow ray itself.

This removes the need to shoot primary rays, massively reducing the amount of rays shot for the scene overall, and because this information is needed in a deferred renderer anyway it costs us nothing extra in terms of computation. It is in fact necessary anyway to perform lighting calculations after the shadow rays have been shot in order to correctly mask the lighting in the shadowed areas. The rendering pipeline is described in greater detail below.

3.2 Ray-Tracing Implementation

This test uses a hierarchical grid structure for ray acceleration. This grid is controlled on the CPU and transferred each frame to the GPU in the form of a texture. In order to encapsulate the ray-tracing part of the program the majority of functionality is kept within a RaySystem class. This class holds a reference to the grid itself and also controls the creation of the shadow masks for each shadow-casting light in the scene. The grid is housed as a one dimensional array of floats which contain the object IDs at that point. There are helper methods to quickly place an objects ID into the correct position in the array and remove an ID from it when necessary. The data in this one dimensional array is copied to a texture each frame and sent to the GPU. This speeds up the creation of the texture but results in slightly more cost when adding or removing an object when compared to a simpler 3D array.

A uniform grid structure was also tested which used a 3D array where the first two

elements were the X and Y coordinates of the node and the third element was an array of the object IDs in that node. This is the more logical way of housing the grid on the CPU however it was relatively slow when copying the data from that array to a one dimensional array that could be copied to a texture. The uniform grid also proved to be quite slow when objects overlapped into neighbouring nodes. In this situation the object must be referenced in each node, this led to the situation where a single ray was testing against the same object multiple times.

Each ray-occluder, that is an object which will create a shadow, calls a method in the RaySystem which places the object ID into the correct node in the grid according to its starting position. Each frame after this the object calls a method to check if its new position is still within the same node, if it is then no changes need to be made, if the position has changed then the ID is removed from the old node and placed into the new node. This method proves to be quite quick and can easily handle several thousand objects.

Each frame, after all the ray-occluders have checked their position, the grid is converted to a texture and transferred to the GPU. Each node in the texture can be seen as a square block which houses the object ID references. For this test the maximum object count per node is set at 256 and each node is 32x32m in size. As many smaller objects do not cast shadows this limit should not be too noticeable. It is of course changeable to any value needed but to achieve more objects per node while keeping the node the same size will require a larger texture. For this test the scene is 2048x2048m in size. To get the necessary texture size given the 32x32m node size and the 256 object per node limit it is simply a matter of doing the following:

$$Tsize = \frac{Ssize}{Nsize} \times Ocount$$

Where Ssize is 2048, Nsize is 32 and Ocount is 16 (256 objects in a square block gives us 16 on each side). This gives us a texture size of 1024x1024 pixels. Any current graphics card should not have a problem handling a texture of even 4096x4096 in the single format used for the object reference.

When considering scenes with high amounts of shadow occluders within a small area it might be best to use less nodes with a higher object count each. This would be better for indoor scenes where there are often dozens of objects in each room. In an urban environment game the scene size could easily be lowered to allow far more objects per node. For example if the scene size was just 512x512m and the nodes were kept at 16x16m then with the same size texture we would be able to have 1024 objects per node, plenty for even the most detailed environment. It would also be feasible to have separate grids for different areas of an urban scene and only update the objects associated with each grid area if the player enters it.

The ray-tracing occurs after the first steps of deferred rendering. The deferred renderer uses rasterisation to draw each objects depth from the camera, diffuse colour, normal vector and specular information to render targets. The 3D position of the pixel can be calculated at later stages by using the back four corners of the cameras view frustum to get the directional vector for each pixel and multiplying that by the depth. In the ray-tracer this 3D position is used as the start of the shadow ray and the direction of the ray is the vector from that point to the light whose shadow mask is being calculated.

Each shadow casting light must have its own mask, for the test scene a limit of four shadow masks is used and only the nearest four lights have their mask calculated. It is possible to force a light to be included in these four, for example in the test scene the directional light for the sun will always have its shadows calculated. For better performance this limit can be lowered or for simpler scenes more shadow masks can be calculated.

The actual ray traversal is based on an optimised version of the DDA algorithm and is gone into more detail in section 2.3. For each node it must test against the ray checks if it is above the nodes maximum height. This max height is calculated on the CPU during the grids creation, each object as it is placed in the grid checks whether its current y position plus the radius of its bounding sphere is greater then the current max height of the grid node, if it is then the grid nodes max height is increased. From this we know that if a rays minimum y position in that node is greater then the nodes

max height we no longer need to test it. There is also an overall scene max height, as soon as the ray goes over that height, and the ray y direction is negative we can discard the ray. If the ray is inside the node it then loops through each object in the node. It does this by going through the nodes list of object IDs then uses those IDs to lookup the object information in the object texture.

The object texture contains information for each object in the scene and is updated every frame. The object data it contains is the objects inverse world matrix, the model ID, the maximum scale and the primitive type. The inverse world matrix is used to transform the ray into model space. It is far quicker to transform the ray to model space then it is to transform each vertex of the model into world space. The model ID is used to first look up the bounding sphere information which is kept in a texture that is only updated when a new model is added to the scene. The bounding spheres radius is multiplied by the maximum scale of the object; the maximum scale is obtained on the CPU by simply taking the maximum element of the objects 3D scale.

The ray then performs an intersection test against the bounding sphere and if that returns a positive result the actual mesh data is retrieved using the model ID from the model information texture. This model information texture is the combined meshes of every model in the scene; as such it has to be very large. A 2048x2048 texture is used in this test that allows for a limit of 4,194,304 total vertices in all the models used. Considering that games use model repetition a lot to avoid too much time spent on content creation this should not be too much of a limiting factor. The other half of the model information is the indices which are held in their own 2048x2048 texture, these indices contain pointers to the vertices in the vertex texture.

For each model the ray tests intersection against it uses the model ID to look up the start and end position in an intermediate model data texture. This can be a small texture the same size as the bounding sphere data texture. All this texture contains are two floats holding the start and end position of the mesh in the index texture.

3.3 Deferred Lighting

Deferred lighting, as described earlier allows us to calculate lighting only for the pixels that will end up in the final image. This allows for far more lights in a scene overall, highly optimised deferred engines are capable of calculating hundreds of point lights in situations where a forward renderer could only manage a dozen. Many game engines are now being converted to use deferred rendering, notably STALKER[30], CryEngine 3[31] and Star Craft 2[32].

Deferred lighting excels at scenes with many local lights, for example a nighttime scene where a lot of people are walking around with flaming torches. However forward lighting can still perform better in some situations, particularly large open scenes with just one directional light for the sun. In general however many game designers are starting to prefer the wider range of possibilities presented by deferred lighting. The reason why it is so much more efficient with local lights is that only the pixels possibly affected by the light have calculations performed. So not only does the method cull any overdrawn pixels it is also possible to easily cull any pixels outside the range of the light source.

This is usually performed by rendering a primitive shape in order to perform the lighting calculations. For a point light a sphere is used, for a spot light a cone is used and for a directional light a box is used. In the pixel shader for these primitives the g-buffer is read and lighting calculations are performed. With just this method objects too far behind the light would still be tested, details of how to avoid this using z-testing and the stencil buffer are provided in a presentation from nVidia[33].

Deferred lighting is also perfect for the integration of ray-traced shadows. It essentially encapsulates the lighting pass, completely removing it from the geometry pass. This allows us to use any information needed from the g-buffer between the two main passes. This simplification of the rendering pipeline is also another reason why it is popular with game developers and game artists alike. It ensures that all objects are treated equally in terms of lighting, only the data passed through in the g-buffer matters. It is also very useful when performing high dynamic range (HDR) lighting. The

light calculated in the lighting pass can be output to a separate texture. Traditionally in a forward renderer the luminance value would have to be calculated afterwards by the values of the red, green and blue channels in the back-buffer. With a separate texture containing the amount of diffuse and specular lighting for each pixel this calculation is no longer needed. This separation of lighting from the rasterisation process leads to a great many other possibilities and makes a lot of things much easier from a technical point of view.

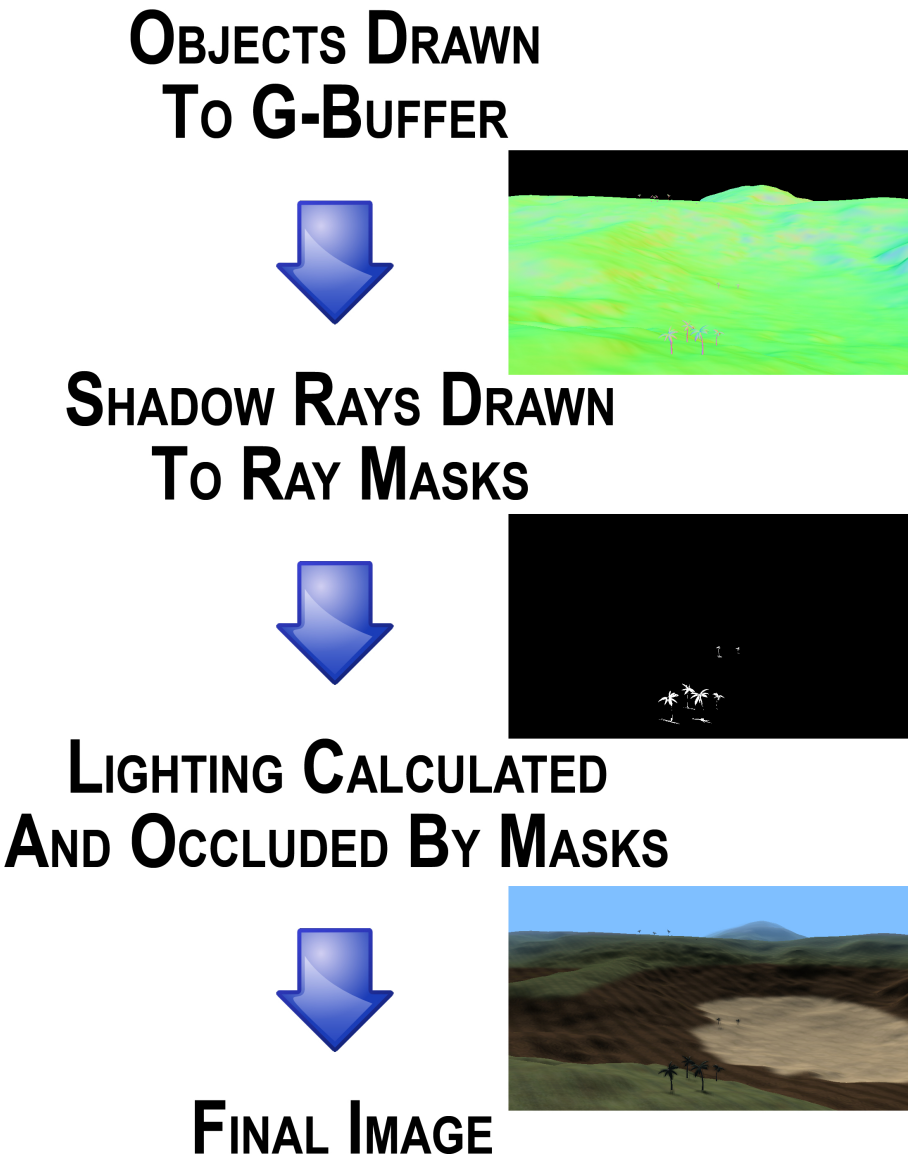

Figure 3.1: The Rendering Pipeline

3.4 Rendering Pipeline

The rendering pipeline used for this test can be seen in figure 3.1. First the objects are cached in a render manager and sorted by material, then by model and then by their world matrices. In this way the minimum amount of state changes and draw calls are

needed. Each time a material changes on the device it requires a state change which can be quite expensive, it is always best to reduce these to as little as possible, the same goes for draw calls and vertex buffer or index buffer changes. When rendering the objects first the material and all its textures are set on the device, then it loops through each model with that material and for each of those models it sets the vertex buffer and index buffers. Then the effect is started and each world matrix is looped through setting the correct matrix and inverse matrix for normal transformation on the effect. After that the draw call is made and then the render manager moves on to the next world matrix.

The terrain is drawn using a quad-tree with cached draw calls much like the render manager uses for the models. A texture blend map is used to allow up to five textures on the terrain, this is accomplished through the use of a 4 channel RGBA texture. The four channels are used for the four primary textures and the remainder is used for the base map. So if a terrain node has a blend texture of 0.2, 0.0, 0.3, 0.0 the remainder would be 0.5. The first, third and base textures would be used. The quad-tree also makes use of geo-mipmapping although just the distance to the nearest point of the bounding box is used as the level of detail (LOD) heuristic. The LOD structure is made up of several layers of simpler and simpler index buffers. The vertex buffer itself is always the same however to reduce memory usage and bandwidth at draw time. To avoid seams each LOD layer has 16 different possible neighbour combinations which reduce the detail along the sides of the node which have lower detail neighbours. This technique maintains visual quality but vastly decreases the time taken to draw the terrains geometry.

For the first phase of rendering the objects depth, specular amount, specular hardness, normals and colour are output to the g-buffer. The layout of the render targets can be seen in figure 3.2, all of these are 32-bit render targets, the first two being standard R8G8B8A8 format and the third being of R32F format. Once the models have been drawn the next phase is shadow calculations. Because of the encapsulation provided with deferred rendering it is possible at this point to substitute in either ray-tracing for the shadows or traditional shadow maps without any significant overhead bar the performance difference between the two. It is also possible to just ignore shad-

ows entirely and clear the shadow masks so that no pixels are shadowed. This ability to switch quickly between shadow techniques makes the process of benchmarking much easier.

Figure 3.2: G-Buffer Layout

Both shadow techniques use the same shadow casting light information, the location of the light, the type of light, the range and other variables needed depending on the light type. This information is passed to whichever shadow technique is currently being used. At this point the next step depends on the current technique and also the type of light. For shadow maps a view/projection matrix is formed from the light information, in the case of a directional sunlight these matrices are built by taking the cameras view frustum corners, converting them into light space and building the view and projection matrices around these to ensure that each point the camera is looking at is inside the shadow map. For a spotlight a simple "lookat" view matrix would be created and a perspective projection matrix based on the angle of the camera would be used. There are several methods for creating shadow maps for point lights including cube maps and paraboloid maps but these are beyond the scope of this dissertation. Once these view/projection matrices are created it is then necessary to completely redraw the scene into a depth map, this can start to get quite expensive with a lot of objects in the scene. Once this is complete a full-screen quad is drawn which takes information from the g-buffer, calculates the pixels 3D position, converts that to light-space given the lights view-projection matrix and then tests whether it's depth from the light is greater then or less then the depth inside the shadow map. If the pixel is further from the light then what is in the shadow map the pixel is in shadow. This is drawn into the shadow mask which can then be sampled later.

When using ray-tracing for shadowing it is not necessary to draw a depth map from the light so we move straight to drawing a full-screen quad to create the shadow mask. The information from the g-buffer is read in, 3D position is calculated and a ray shot from that point in the direction of the light source. The ray then traverses the hierarchical uniform grid testing each node that it intersects with. The ray-tracing methods used are gone into in more detail earlier in this chapter. The result is a mask which is similar to the one produced by the shadow map method above. The fact that both methods have the same output allows us to switch between them without changing the lighting calculation code in any way.

After the shadow masks are created we move onto the lighting calculations, the calculations themselves are standard NdotL equations with separate methods for directional, point and spot lights. All the information needed is provided for in the g-buffer. For this test the calculations are done for every pixel that is not the background. There are many ways to optimise lighting in a deferred renderer but they are outside the scope of this project. The most popular of these, primitive shapes, is described in the section on deferred lighting. In this particular project the shadow masks could also be used as a pre-check before lighting calculations so that shaded pixels do not waste calculations. This is the last stage of the pipeline and once the lighting calculations are finished the end result is drawn to the screen. It would be at this point that any HDR, tone-mapping or gamma correction methods would be integrated but these are not necessary in this project.

3.5 Ray-Tracing Compromises

As the complexity of the ray-tracer in this application was increased it became clear that to ensure smoother framerates some compromises would have to be made. This was always an option considered from the start however early test results proved that with the rays testing just the node they were in framerates were maintained around 70-80fps without any compromises. Of course this is not a feasible solution for game shadows as the shadows from objects in one node would not be cast into neighbouring nodes.

When the DDA traversal algorithm was incorporated into the ray-tracing process framerates dropped significantly, with any reasonable amount of objects in the scene the framerates proved too low to be playable. It even got to the point where if too much work was attempted by the ray-tracer the GPU driver would fail and crash the application. It was clear that some sort of performance boost was needed.

3.5.1 Down-Sampling the Shadow Mask

The most obvious way of increasing performance in a ray-tracer is to lower the amount of rays that need to be shot. Luckily in our rendering pipeline this is possible through simply downsizing the render target of the shadow mask used by the ray-tracer to shoot the rays in the scene. It was decided to use three different quality settings, the lowest setting divides the width and height of the screen size by four to determine the size of the render target. The medium setting divides it by two and the high quality setting uses the actual size, not downsampling at all. This results in 1/16th the amount of rays being shot for the low quality setting and 1/4 the amount of rays being shot for the medium setting.

This downsampling led to predictable results. In a case where the framerate at the lowest quality setting was 70fps the framerate would drop to roughly 17fps for the medium quality setting and around 5fps for the highest quality setting. These results matched perfectly with the difference in the amount of rays being shot. Without this reduced amount of rays to be shot the results achieved in our scene tests would not have been possible. For the tests the lowest quality of ray shadow was used.

However this does lead to quite noticeably blocky shadows, exactly the type of artefact we were seeking to avoid by using ray-tracing in the first place. This is sadly unavoidable with current hardware. Perhaps with a more optimised GPU ray-tracer the medium setting would be able to be used. This results in far less noticeable blockiness and with Percentage Closer Filtering(PCF) sampling of the resulting shadow mask when sampling for the lighting there is almost no discernible difference between the medium and the high quality settings. With further improvements in hardware in the

coming years it should be easily possible to ray-trace shadows without using downsampling however for more complex, expensive ray-traced effects like reflection and refraction downsampling by a factor of two like our medium quality shadows should still remain an attractive option in terms of increasing performance with minimal effect on the final image.

3.5.2 Ray Interlacing

Another way of reducing the amount of rays that have to be shot each frame is to use ray-interlacing. To accomplish this our application creates a mask the same size as the ray-tracing render target. This mask is a standard four-channel RGBA texture. To create it each pixel is randomly assigned to one of the channels so that each pixel in the final mask is either coloured red, green, blue or alpha. This mask is then supplied to a shader that along with the current frames colour sets the appropriate stencil mask. Each pixel looks up its own value and tests it against the frame colour, if it is the same colour it sets the stencil mask to 1, if it is not then the stencil mask is left at 0. When performing the ray-tracing itself the stencil test automatically checks the mask and ensures that only pixels that have been selected by the interlacing shader perform ray tests that frame. For pixels that do not shoot that frame their value in the shadow mask remains the same from the previous frame. This is achieved using the "RenderTargetUsage.PreserveContents" flag when creating the render target. This flag ensures that the masks calculated in previous frames are not discarded but only overridden when new tests for that pixel are performed.

The downside to using this interlacing method is a ghosting effect when the camera or the object is moving quickly. The shadow results from the previous frames for that screenspace pixel will be incorrect leading to a speckling at the edges of the shadows. In our application the sample repeats every four frames, at a high framerate of around 60fps this speckling is almost unnoticeable as the longest any incorrect shadow sample will remain is 1/15th of a second. However when framerates drop too low, under 30fps for instance the speckling becomes very noticeable and disconcerting. This is lessened if a higher quality for the ray shadows is used but that just causes performance to suffer worse then when the ray interlacing was not used.

Ray interlacing is left disabled by default in our application for our tests. It is a method that may prove useful in the future when improving already high performance but in a situation like ours where the framerate will be low anyway the end result is not worth the few frames per second performance benefit it shows.

3.5.3 Limiting the Distance of Rays

So far we have discussed two ways of limiting the amount of rays shot in the scene per frame, this next method is both a way of reducing again the amount of rays shot and also a way of shortening the amount of time each ray takes to calculate once it is shot. First of all we must remember that this ray-tracing method is aimed at a game environment, we need only concentrate our visual quality on what the player himself is concentrating on. In many games reducing the distance from the camera at which shadows are drawn is a good way of ensuring more stable performance and better quality shadows. Indeed for our shadow buffer implementation we use this same limiting technique to reduce the area over which the shadow map must stretch, thereby increasing the detail provided in the area closer to the camera.

It is only fair then that the ray-tracing technique should also benefit from such an optimisation. In the ray-tracing shader we already sample the depth of the pixel to determine the world space position. It is simply a matter of, after sampling the depth, checking whether the depth is greater then whatever limit we have in place. If it is then we can simply discard that pixel and move on. We also use the depth test to ensure that the depth is greater then 0. A depth of 0 would indicate that it is a background pixel and as such did not get assigned a depth value during the rasterisation phase. This effectively reduces the number of rays that have to be shot to only those within the visual attention area of the player which is usually in the foreground of the image during gameplay. For our application the limit at which shadows are drawn for both the ray-tracing and the shadow buffer method is determined by using half of the current draw distance. This allows for further scaling in performance with shorter draw distances meaning less rays to shoot. For our test scenes we used a draw distance of 1024 metres which gave us a maximum shadow depth of 512 metres.

The maximum shadow depth is also used during the rays traversal. It is integrated into the DDA traversal loop so that if the distance the ray has travelled exceeds the value then the loop will end and the ray will return negative. The loop also ends if an intersection is found or if it goes outside the map bounds. This can lead to some irregularities where distant large objects do not cast their shadows further then the maximum value but in our test scenes there was never a time when an object would cast its rays anywhere near the 512 metre limit.

During testing using the shortest draw distance of 128 metres, giving us a shadow range of 64 metres, the resulting benchmark was over twice as fast as the ones using the furthest draw distance. This shows the power that controlling the draw distances has over performance in a game engine. There were however still dips down to around 15-20fps at times when clusters of objects where approached. This is due to the fact that even though there were a lot less rays to shoot, those that were there all had to travel though busy nodes.

Chapter 4

Test Setup

4.1 Hardware Used

The hardware used for these tests is:

- Intel e6300 CPU @ 3.15ghz
- 4gb Geil DDR2 RAM @ 900mhz
- Asus P5Q Motherboard
- nVidia GTX 470 GPU

The operating system used is Microsoft's Windows 7, the main programming framework is XNA 3.1 and the shader language used is DirectX 9.0c. As the main limitation in terms of performance for this test is the GPU let us look more closely at the specification of it.

- GPU Engine Specs:
  - CUDA Cores: 448
  - Graphics Clock: 607 MHz
  - Processor Clock: 1215 MHz
  - Texture Fill Rate: 34 billion/sec

- Memory Specs:
  - Memory Clock: 1674
  - Standard Memory Config: 1280 MB GDDR5
  - Memory Interface Width: 320-bit
  - Memory Bandwidth: 133.9GB/sec

These specifications have been taken from nVidia's official GTX 470 product page[34].

4.2 Test Scene

A large open environment with a sizable terrain heightmap is used for the test. A physics engine is incorporated for dynamic objects and standard lighting techniques such as specular and normal maps are calculated in order to match what happens in a game engine. All moving objects in the scene that do not have physics based movements have recorded paths in order to ensure greater similarity between tests. The physics engine uses the elapsed time from the previous frame for integration. As the benchmarks are time based this should result in the the same results each time the benchmark is performed.

The camera also uses a pre-determined path which takes it through areas of both high and low object density. The differences in computation time between the shadowing methods as the camera goes through these different areas should be visible on the resulting graphs. One of the main areas where the ray-traced shadow method benefits is when the camera is looking mostly at the sky or when the camera is close to the light and the rays do not need to travel far through the object grid before they reach the light. Alternatively the shadow-buffer method works well when there are not many objects near the light which have to be rendered into the buffer.

In order to rule out any differences between the tests a sample of five timed runs per test are calculated and the average of these taken as the overall result. This is to cancel any inaccuracies caused by outside effects whether they come from hardware or software issues. Several different scens are tested in order not to favour either of the

shadowing methods by staying, for example in an area with low object count which would favour shadow buffers.

The scene also has models of varying complexity in terms of triangle count. These additional triangles should stress the ray-tracer significantly, far more than the shadow buffer method which uses traditional rasterisation methods which have the benefit of working extremely quickly on modern GPUs. Even on modern high-end GPUs it has been found that having more then a few objects of over 1,000 triangles in close proximity causes quite a substantial drop in performance.

4.3 Physics

The physics engine used for the test setup is JigLibX [35], this is a port of the physics engine JigLib [36] for XNA. JigLibX is quite popular among many XNA game designers for it's robustness and ease of integration into an existing project. The engine supports many primitive shapes as well as a heightmap. For the test setup a heightmap is used for the terrain and several objects of different shape are placed or dropped into the scene to emulate the physics workload in a usual game engine.

It is important to try and match the background resource usage in order to produce a result comparable to modern game engines. One of the most demanding parts of a game engine is its physics engine. One notable example is Mafia II [37] from 2K Czech. This game uses nVidias physX physics engine [38] which is largely based around CUDA, their GPGPU programming framework. When the game was first released it was found to be almost unplayable with physics settings set to high. It was soon discovered however that disabling just the cloth effects on characters clothing fixed the issue.

It can be seen from this that graphics are no longer the only concern when it comes to resource usage in a game, with processing speeds increasing it is now possible for stunning realtime physics effects to be introduced into games like cloth, damage systems and fluid simulations. Because of this it can no longer be acceptable to take just a graphical demo and assume that because this demo can run in realtime that it is suitable for a large scale game engine. This has been the situation with much of the

research into realtime ray-tracing in recent years. While the researchers themselves might understand this the general public see these ray-tracing results and expect to see such quality in their next game purchase.

While JigLibX might only support basic rigid-body collisions it should to some extent atleast mimic the physics engines in most games today. It would be interesting, but outside the scope of this paper to examine fully the balance of resource use across the various systems in a modern game engine to see just how much frame time is actually given over to rendering. This could then help fairly represent the demos of purely graphical effects.

4.4 Scripting

Scripting is a very important part of any benchmarking process. It allows for a wider array of dynamic scenes that also are repeatable in order to produce comparable results when using different rendering methods. Essentially scripting is a list of commands that tell the game what to do and when. These lists of commands are kept in an external file, usually a plain text document easily editable outside the engine code. This removes the necessity to hardcode the placement of objects in every scene. In more extensive scripting environments it is possible to define intricate behaviours from AI to character control and story sequences in scripts alone. Such elaborate techniques are outside the scope of this dissertation though.

In our scenes a custom scripting language is used to set up the starting entities, add the necessary components to them which define their behaviour and then throughout each benchmark to introduce or delete objects as well as control the camera and other objects movement along their respective paths. Code sample 4.1 shows an example of a short script that adds a new entity to the scene by loading it from it's appropriate XML file "palm" and then adds a shadow occluder component to it with the primitive type of "Mesh". A clone of this object is then created at a slightly different position and then a timed event is created that adds a new clone after 4 seconds. The -1 in the Y component of the objects positions lets the program know that the height of the terrain at that X/Z point should be used instead of a specified Y position.

```
1  AddEntity(palm; Vector3(1000.5, -1, 1024.4))
2  AddComponent(palm; Cmp_ShadowOccluder; Mesh))
3  AddClone(palm; Vector3(1024.5, -1, 1024.4))
4  Timed(4000.0; AddClone(palm; Vector3(1050.5, -1, 1024.4)))
```

Listing 4.1: Custom Scripting Language Example

Another benefit of external scripting is the ability to record actions and object placements in a script file and then reproduce them exactly. For example it is possible to load an empty scene then go through it placing objects wherever the user desires. Once done the actions the user performed can be saved either to be performed immediately upon loading a scene or at the specific times at which they were created by the user.

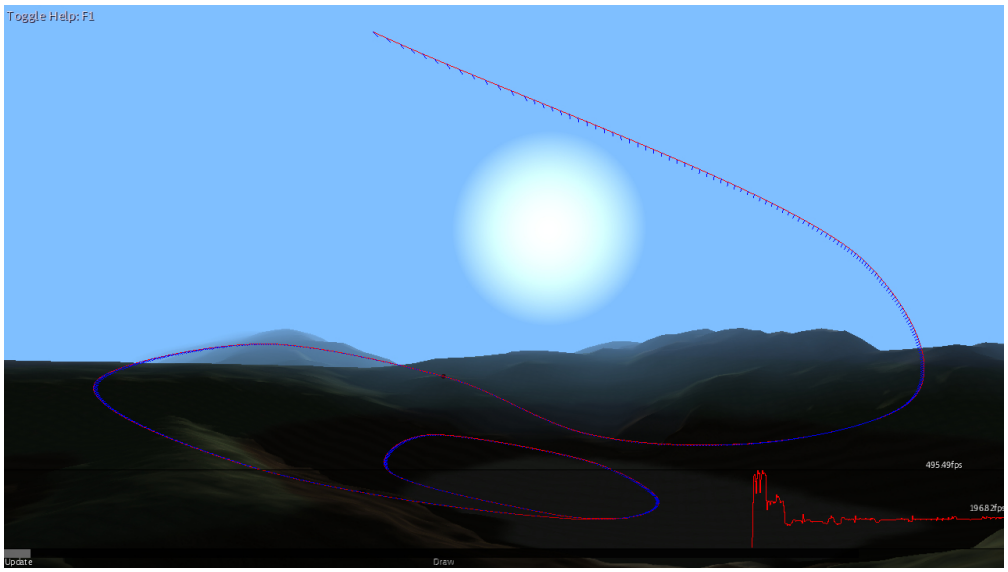

Figure 4.1: Screenshot of a path.

Figure 4.1 shows an example of a path recorded as an XML file and loaded into the engine as a set of points for a sphere in this example to follow. The path itself can be seen as a red line and the direction the rotation of the object is shown as blue lines along the path. To blend between waypoints in the path the position calculations use a Catmull-Rom formula which takes in the two previous and two next points to create a smooth curve between them. The rotations are similarly blended using spherical interpolation of the quaternions at each point.

Bringing all these options together allows us to create sufficiently dynamic yet re-

peatable scenes with which to perform our benchmarking. They allow us to set a path for the camera to follow through both sparsely and densely populated areas as well as have areas of high dynamic object movement and scenes with a vast amount of physics based objects in order to test the impact of other high computation processes on the different shadowing methods. It is necessary to test such an array of different situations in order to get relevant results as most 3D games have a similarly broad range of locales.

4.5 Performance Recording

Each scene is run three times per benchmark. Once for no shadows as a control sample, once with buffer shadows and once with ray-traced shadows. Each benchmark is then performed five times with results being recorded to a text file each time. The results are recorded by a dedicated class which gets called to draw a framerate graph to the screen each frame. It records the framerate by dividing 1000 by the current elapsed milliseconds to find the amount of frames per second the program is currently running at. It then records this to a temporary list. After a certain interval length, 250ms in the case of these benchmarks the average of the results in the temporary buffer are taken and recorded to a permanent list. This ensures that the amount of recorded framerate samples is the same whether the current framerate is 5fps or 50fps. It can rarely lose a record or two if there are extended periods under 4fps but this is not usually the case and does not effect the end results significantly. It is possible to merely trim a recorded framerate from the end of the other saved results when compiling all the results together.

At the end of each run through the scene, before moving on to the next shadow type the record is stored in a text file and the graph reset for the next test. Each pass of each benchmark is recorded like this in its own text file which can be easily copied into a program like Microsoft's Office Excel where the different results from the five benchmarks can be averaged and the results of each shadow type compared against eachother. there are a host of different methods of comparing performance in a situation like this.

4.6 Comparison Methods

The first comparison method used in this test is a simple graph of the averaged framerate throughout the pass through the scene by the camera. The camera moves at a constant speed so each test will be almost identical in the time taken to reach the end so the results of each run are comparable against eachother. This will show us which methods are slowest and more importantly at what points in the scene are they the slowest. This should allow us to gauge what exactly is causing the slowdown in each case. The second comparison we will be looking at is the total maximum, average and minimum framerate of each of the three methods. This is the average throughout the entire pass through the scene, we simply take each result passed through to the graph previously and find the average of them all. This shows us which method is fastest overall and by how much exactly.

Measuring using just the framerate of the application is not as straight-forward as it would seem. Framerate performance is not linear, for example if we take a program running at 100fps and half the performance it predictably drops by 50fps. If we half the performance again however it only drops by 25fps. Essentially as the time taken to render a frame goes up, the effect this has on the framerate goes down. If a frame takes 20ms we are running at a framerate of 50fps. If we increase the time to 40ms we are running at 25fps. If we increase it by 20 again to 60ms we are running at 16.66fps. It is often better therefore to look not at the framerate but at the time taken each frame.

In order to do this it is not necessary to record a separate list of results for each frame, we can simply divide 1000 by the framerate at that point and we have the amount of time taken to achieve that framerate. This can be done using Excel and the results we already have. Our next comparison method then is the additional time taken over the control set in order to calculate the shadows for each technique. Removing the amount of time taken by the control set allows us to clearly see the exact impact on the frame calculation time that each particular method is having. This is against presented both in a graph of the results over the length in time of the scene and also as a chart showing the total average time taken by each method.

Frames per second and the amount of time taken to calculate the frame, usually in milliseconds, are the two most common ways of measuring studies like this. However in a game environment there are other issues as well. One of the foremost being the overall stability of the framerate, there are a few ways we can study this. Firstly we can look at the difference between each of the benchmark tests. Once we have the average from all ten tests we can find the difference of each one from the average and add them together. Next we divide by the average framerate to get the difference relative to the current framerate. So if we are on 20fps and the difference is 5fps up or down we assume that it is more noticeable then a 5fps difference at 40fps.

This should be an accurate way of gauging the effect on gameplay that such erraticness would have. These differences can then be portrayed as a graph similar to the original framerate graph. This shows us where the major inconsistencies are through the course of the cameras passing through the scene and we can then try to work out why that is. For an additional visual view of the erraticness each of the ten results can be plotted on a graph so that we can see where performance has dropped or increased between the different tests. Often times this is due to an external event such as another process taking up clock cycles on the computer. Such erroneous results can generally be ignored, however long-term erraticness is something that will effect gameplay and must be examined.

The next method of determining the stability of the framerate is to perform what is known as a standard deviation test on our results. For each value we have we subtract the overall average from it, then multiply that result by itself to ensure it is positive. We then add together all these squared values then divide that result by the amount of values - 1. Luckily Excel has an inbuilt function to do that for us called 'STDEV'. Again to get this result relative to the current framerate we divide it by the average framerate. This is portrayed in a chart for the three different tests performed.

With these different comparison methods it should be possible to accurately determine the performance difference between the different methods, the total time taken for each different shadowing method and also the impact this will have on gameplay particularly when it comes to erratic framerates. This is something that game design-

ers constantly try to avoid. It is pointless to have a game that runs at 100fps 90% of the time but in some areas of the map the framerate drops to 10%. This is something that will damage the reputation of the game and the game players experience.

Chapter 5

The Results

In this chapter we will be examining the results presented to us by the tests described in the previous chapters. Results are compiled in several different ways, firstly we shall look at the issue of framerate as this is the method most commonly used when benchmarking a games performance. After that we will look at the average time taken per frame for the different shadowing methods as this will give us a more applicable level of performance difference. Some other factors will also be looked into such as the stability of the performance throughout the benchmarks and what sort of issues affect each method. Due to the lack of significant research in this area it is hard to find other results to compare our own to, this would also be quite pointless when using completely different hardware.

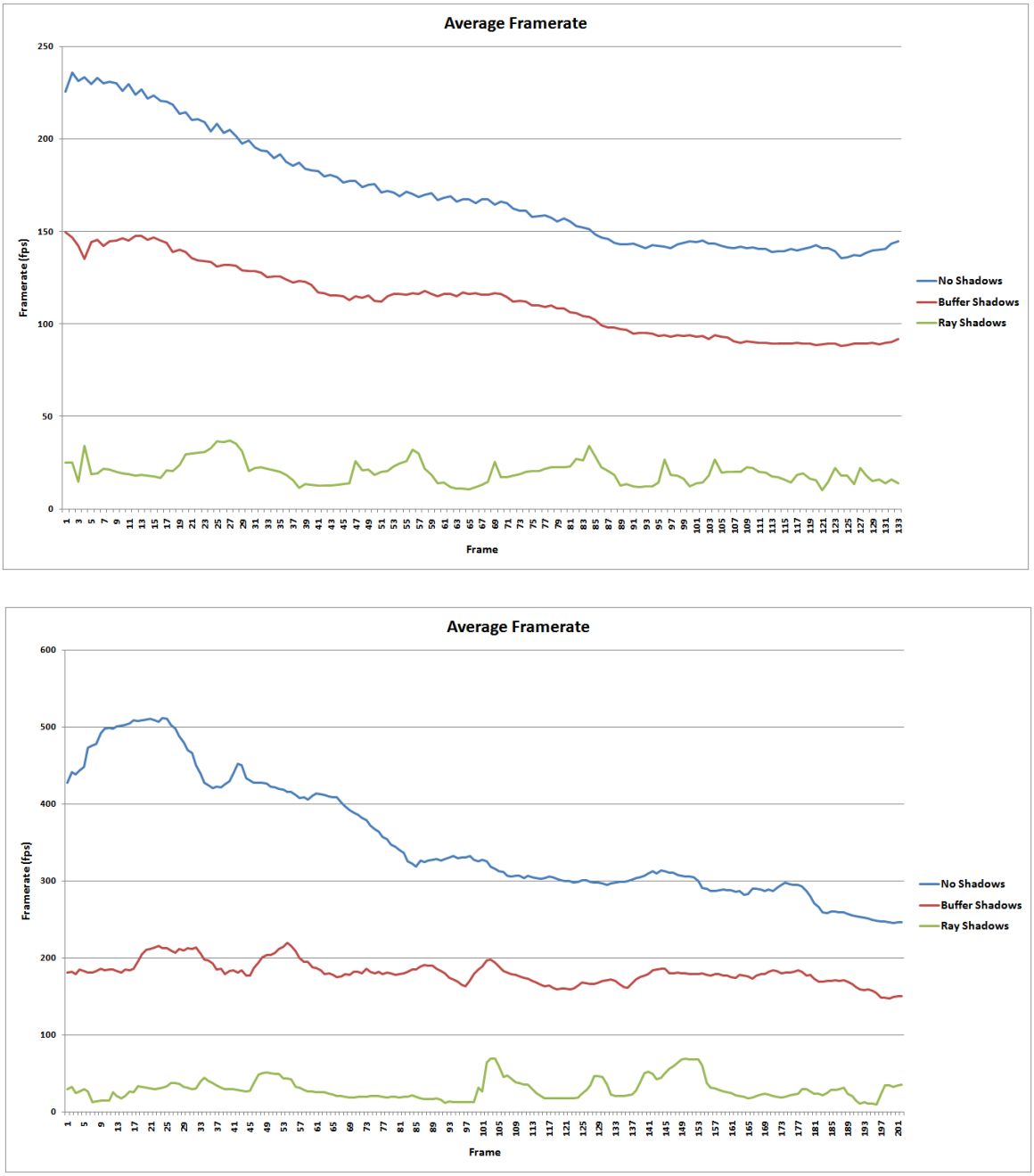

Figure 5.1: Average Framerate Graph for Scenes 1 and 2

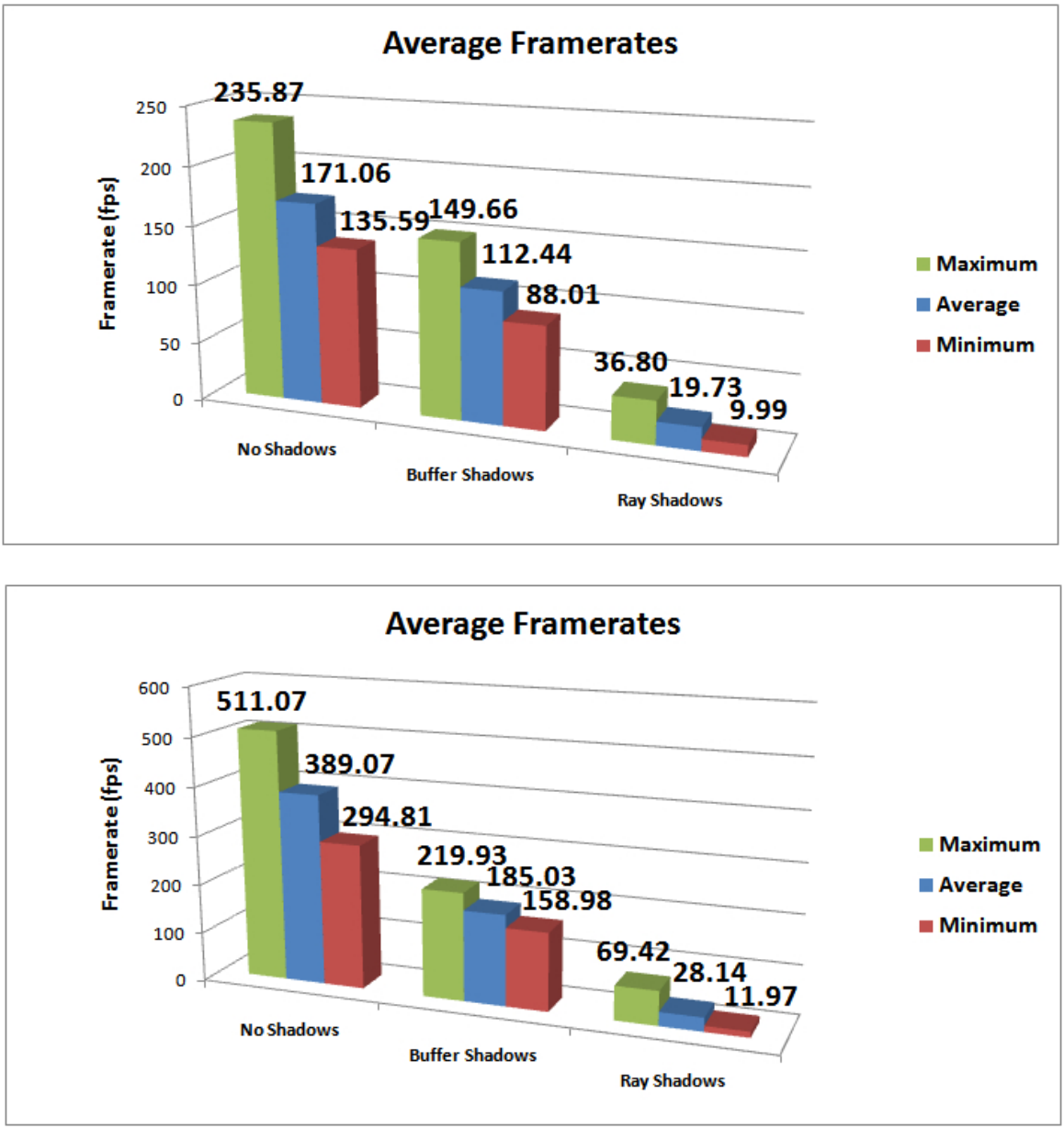

Figure 5.2: Average Framerates for Scenes 1 and 2

5.1 Average Frame Rates

The average frame rates for each method in each scene are determined by running the benchmark five times and taking the average of these to create a graph of the framerate over the time of the benchmark. We find the overall average framerate from these. We use the average of five tests in order to nullify any outside inconsistencies between the tests as well as some of the erraticness inherent with this sort of testing.

If we look at figure 5.1 we can see the average framerate graph for scene 1. This scene has the camera traveling through areas of dense object count as well as a lot of physically dynamic objects and constantly spawning objects near or in front of the camera as it travels along the path as well as many static relatively high poly-count trees. The blue line is the results without using any shadowing method, the hierarchical grid is still updated but tests showed that removing these updates results in no obvious improvements in framerate. We can see as the framerate drops towards the end of the scene the effect of the increased count of objects and the increased physics calculations needed. This is predictable behaviour and gives us a good control sample with which to test our two shadowing methods against.

The red line shows the framerates of the buffer shadow method used. This is quite a stable line dropping towards the end for the same reasons as the control sample. One thing of interest to note is the lesser drop overall between the start of the frame and the end. While the increased objects still have an effect its result on the framerate is lessened. This is due to the shadowing computations taking up so much time that the effect of rendering the new objects and calculating the physics of each is lessened relative to the overall frame time. For example if the frametime is 4.3ms for the control sample and the physics take for example start at 1ms the drop will be more significant as the physics computation time increases then if the frametime for the buffer maps is 10ms.

This masking of the cost of the new objects reaches the point where it is almost unnoticeable in the case of the ray-traced shadows where at the same point in time the frame takes 50ms to render. 1ms of physics calculations either way will really not

be noticeable as is proved by the resulting graph. Another point of importance is the erraticness of the frame count in the ray-traced results. This is due to ray-tracing being affected far more by many different factors of the current frame, for example how much of the scene is sky and so does not need rays shot for it. Of even greater effect is the amount of objects each ray in the image need to test against. If there are a lot of objects in the rays starting node or the nodes in the direction of the sun we can see framerates drop significantly.

In figure 5.1 we can also see the resulting framerate graph for scene 2. Scene 2 has less objects overall and groups these objects into tighter clusters then scene 1. As before there are a mix of static and dynamic objects. There are two things of interest to examine regarding the difference between this scene and the first scene. Firstly is how well the different shadowing methods scale when there are less models in the scene and secondly will be what sort of effect does the clustering of objects have on each method. Straight away we can see the effect clustering has on both the shadowing methods, this effect is less obvious, and even non-existent in some places for the control sample however.

Because the shadow buffer method relies on a view-projection matrix created from the cameras view frustum when drawing the shadow buffers depth only those objects visible will be drawn. So when the camera comes to a cluster of objects, not only do they add time in the initial g-buffer rendering they also add time to the shadow buffer rendering too. When the camera is not pointing at those objects in the cluster there is a good chance that they will not have to be rendered by the shadow buffer either. Of course this is not always true, especially in the case where objects in the direction of the light source cast shadows back onto the view frustum of the camera even though they are not directly visible themselves.

The ray-traced shadow method is even more affected by the clustering of objects, large dips and rises in the framerate can be seen in roughly similar areas to those seen in the shadow buffer method. This is due to the screens secondary rays having to test against nodes with far higher concentrations of shadow occluders. In some areas with very few objects the framerates rise to 60 or 70fps. It is clear from this that the layout

of the scene is very important when it comes to ray-tracing, an even distribution of objects throughout the scene is far more efficient then a clustered approach.

5.2 Total Average Framerate

In figure 5.2 we can see the total maximum, average and minimum framerates for the entire benchmark of scene 1. Predictably the expensive ray-tracing method comes last in terms of performance with an average of just 11.53% the performance of the control sample. The shadow buffer method has an average performance of 65.73% of the control sample making it almost 6 times quicker then the ray-traced method in terms of average framerate. For this scene with an even distribution of shadow occluders the maximum, average and minimum distribution is as expected. The minimum framerates are arguably the most important values given here. The buffer shadow method will clearly give perfectly smooth gameplay with framerates never dropping below 88fps however the ray-traced shadow method, while the average could be worse, the minimum framerate would criple the gameplay leading to it being unusable in this form on current hardware.

Beneath this are the same set of results for the second scene. Overall the performance is much better then the first scene with most figures one and a half times, if not more the equivalent figures in scene 1. The most striking difference is in the maximum framerate of the ray-traced shadows. Here we can see the impact of areas with little or no shadow occluders to test between the shadow start point and the light source. This shows us the speed of the traversal algorithm when there are few object intersection tests to perform. With most LCD monitors operating at 60hz today any framerate above this results in the same smooth gameplay. Of course this is irrelevant as the entire object of this test was to see how ray-traced shadows would fare in a normal game environment.

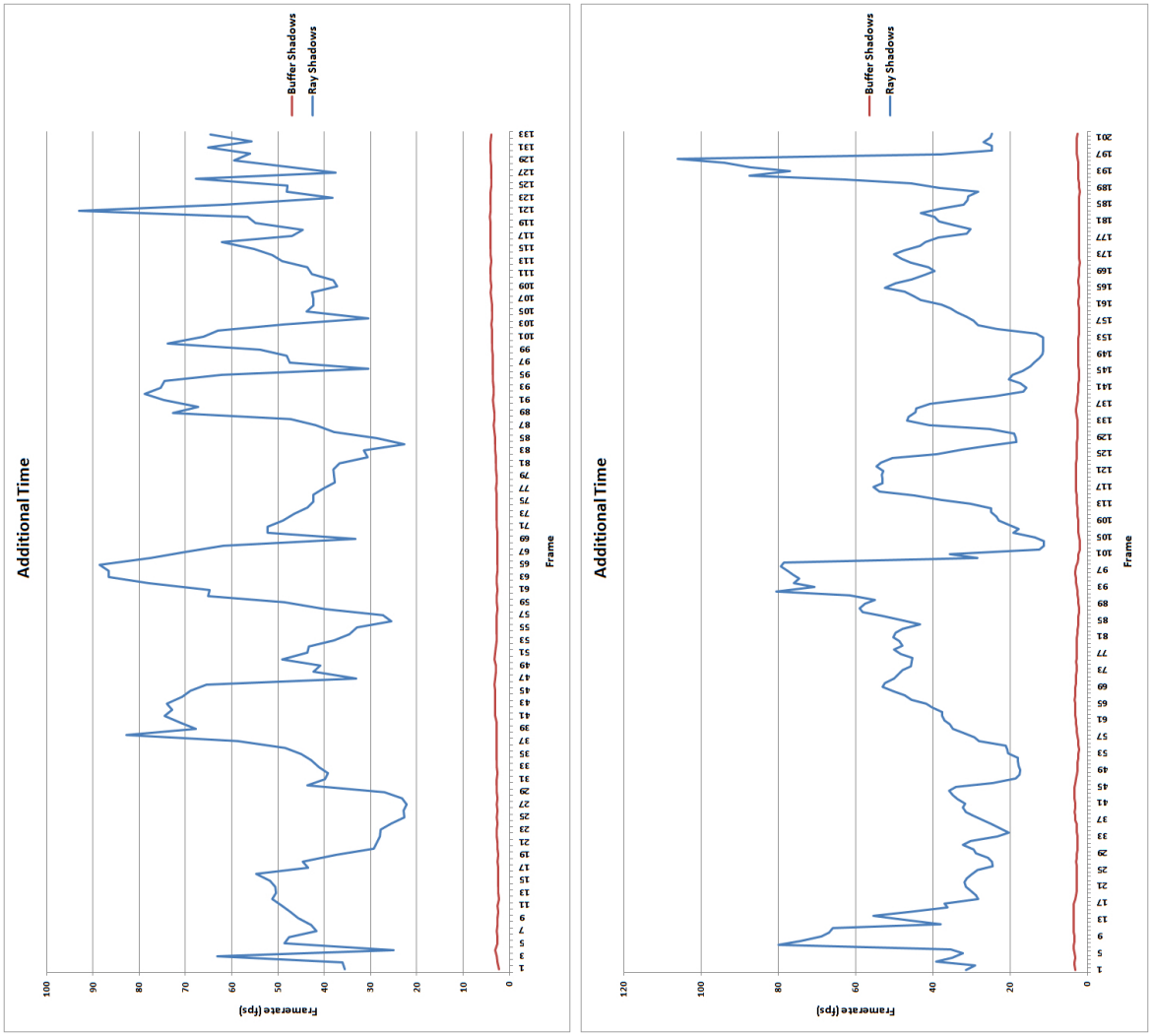

Figure 5.3: Average Frame Time Graph for Scenes 1 and 2

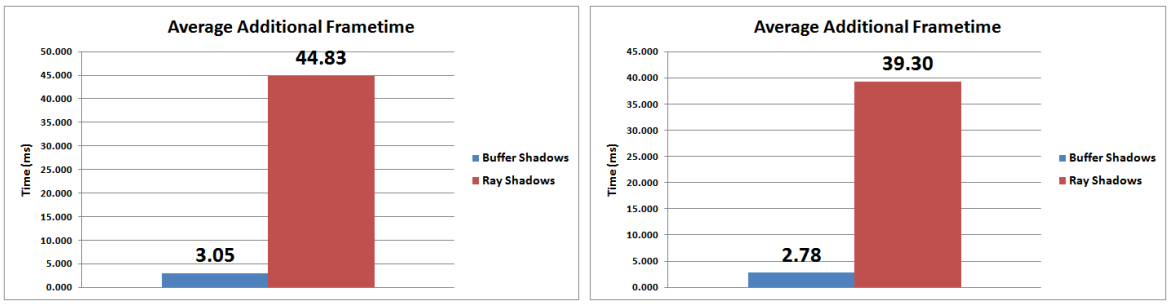

Figure 5.4: Average Frame Time for Scenes 1 and 2

5.3 Frame Calculation Time

As described in the Test Setup chapter a more applicable way of measuring performance in a situation such as this is to measure the amount of time taken per frame in addition to the control sample that is needed to calculate the specific shadowing method. In figure 5.3 we can see the amount of time taken per frame over the course of the benchmark. Again, in the second graph, we can see the effect of clustering on the time taken with big dips in the amount of time needed in the areas of sparse object coverage. We can also see a significant drop from the first scene to the second scene, these results match those of the framerate graph earlier in this chapter.

If we look at the average additional frametimes over the entire course of the benchmarks we can see that the ray-tracing method takes over ten times longer to computer then the shadow buffer method. In scene 1 with a lot of objects in the scene the ray-tracing method takes 14.69 times as long to computer. In scene 2 with fewer objects in tight clusters the difference is 11.62 times as long. The reason for the difference in performance comparisons between the two methods could possibly be down to the inclusion of a lot of primitives in the form of spheres in the second scene. Rasterisation does not handle spheres well, instead of treating them as a primitive in their own right they must be broken down into triangles, to form a smooth sphere this can require thousands of polygons. However when ray-tracing it is possible to use the equation of the sphere to test intersections against it. This is one of the few areas where ray-tracing is actually faster then rasterisation.

The nVidia GTX 470 is capable of an estimated 1,088,640,000 floating operations per second, just over 1 teraflop. To put this into context Intel's ASCI Red supercomputer built in 1997 was capable of 1.34 teraflops[39], exactly the same as nVidias GTX 480 GPU. The difference between the ASCI Red and the GTX 470 or 480 is that it took up 2500 square feet, used 9298 Pentium processors and required 850kW of power. In just 13 years the same raw processing power is now available for a few hundred euro in something the same size as a book. If the same speed of development continues we can only hope that there will be no problem, in several years time handling the demands of a realtime ray-tracer.

To come back to the time taken on todays hardware however we can calculate, very roughly the amount of floating point operations required for each particular shadow method. If we take the results from the first scene of an average of 112.44fps for buffer shadows and 19.73fps for ray-traced shadows we can divide the 1.088 teraflops by these figures to find that buffer shadows require 9.68 billion floating point operations on average per frame and ray-traced shadows requires 55.14 billion operations per frame. These are very rough figures assuming perfect 100% usage of the GPU at all times, this is of course not true in any situation but they give us a basis of measurement to use against other hardware in the future. For example, to estimate the power needed by a GPU to achieve an average of 30fps with the ray-traced method we can just multiply the 55.14 billion op/frame by 30 and find that we would need a GPU capable of atleast 1.65 teraflops. Assuming Moore's Law holds true this should be achieved in the next iteration of graphics cards, in fact ATi currently claim to achieve over 2 teraflops in their current generation of graphics cards but in terms of performance even their best card is very similar to nVidia's GTX 480 so it is not clear how they calculated their figures. In order to ensure minimum framerates never drop below 30fps we would need a GPU capable of over 3.27 teraflops. This is of course with the current implementation which I am sure could be far more highly optimised especially if a dedicated GPGPU programming language like OpenCL were to be used. This will be gone into further in the Conclusions chapter.

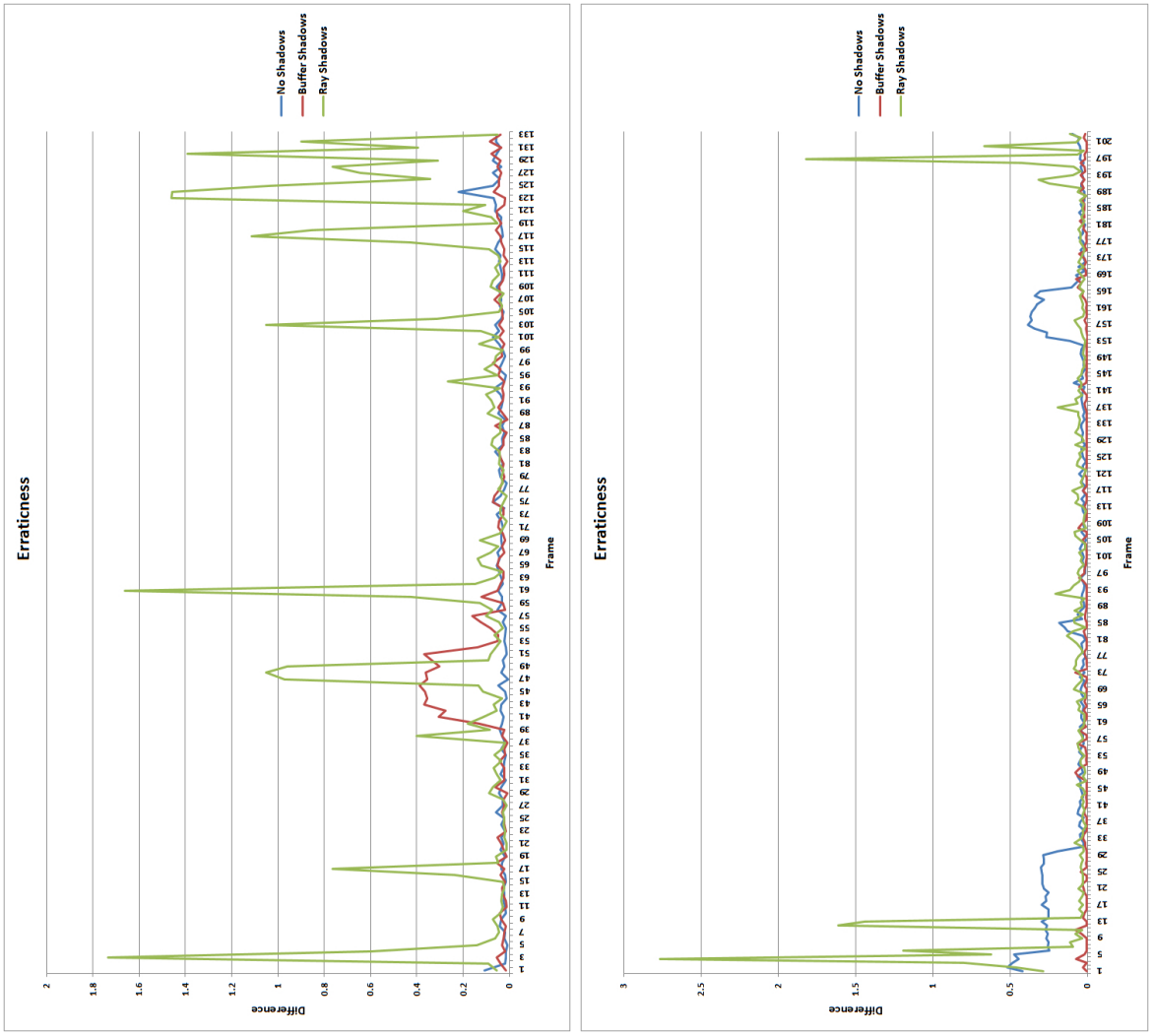

Figure 5.5: Erraticness Graph for Scenes 1 and 2

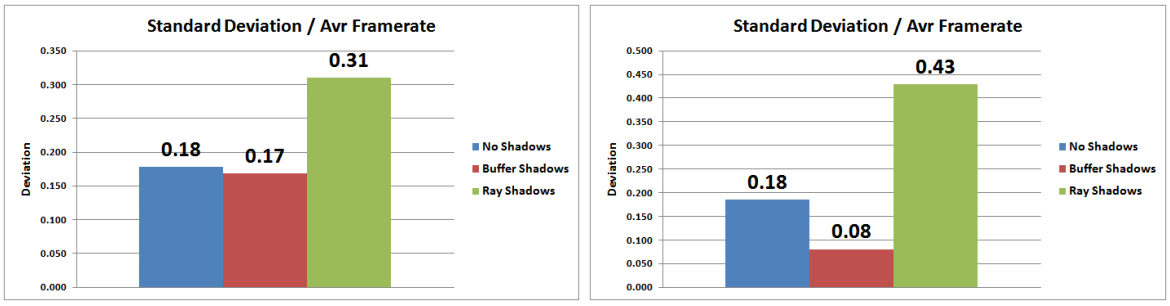

Figure 5.6: Standard Deviation of Frame Rates in Scenes 1 and 2

5.4 Framerate Stability

A good engine relies not just on achieving high framerates, or even high average framerates, it must also be capable of achieving a steady framerate. Having constant dips in speed can be more disconcerting then having a stable, if slower framerate. A lot of this is down, not just to the engine designers but also to the level designers and artists using the engine. If you were to examine most high quality games on the market today you would see that in areas with large highly detailed objects like huge buildings or statues an effort is made to ensure that the nearby surrounding locality is comparatively low detailed.

In order to test the stability of the framerate we can look at the erraticness graphs in figure 5.5. This takes the overall difference from each of the samples from the median and divides it by the average framerate. This effectively shows us areas in the scene where the framerate varies a great deal. In the erraticness graph for the first scene we can see quite a few spikes throughout the scene. It would appear that these occur in areas where a lot of objects are on the screen at once. The erraticness graph merely shows us the difference between the different test samples, this helps us identify areas of sudden change in framerate as each sample will differ very slightly.

Additionally we have recorded an overall standard deviation from the median for all the samples in each test. These figures in 5.6 again show us how much each of the three benchmarks differ sample by sample overall. It is a good way of seeing which method is the most stable. Oddly, it would seem, the method without shadows shows greater deviance then the buffer shadows method. At first this would seem to be illogical as in the framerate graph the shadow buffer method would appear to differ more over the course of the benchmark. However when examined closer the benchmark without shadows actually has steeper gradients which lead to the greater standard deviation. The reason for this is actually quite simple, when rendering the shadow map itself the light frustum sees further ahead of the camera frustum a lot of times, this means that as clusters of objects approach the shadow buffer is the first portion to have its workload increase, this serves to actually smooth the gradient of the graph when compared to just the control sample. Of course the ray-traced shadows do not have this benefit as

they have no shadow buffer, instead whatever sudden effect of clusters that we see in the method without shadows is amplified by the need to test ray intersections in these new dense areas. This is evidenced by the numbers in the standard deviation figures where the ray-traced shadow methods are three or four times as deviant as either of the other two benchmarks.

Alternatively, instead of relying on the deviance we can of course just look at the framerate and time graphs to see where sudden changes occur. The frametime graphs in particular are useful for this as there are many sharp peaks and troughs for the ray-traced benchmarks which is evidence of the differing workloads throughout each of the two scenes. When viewed in person these sudden changes prove to be distracting as in one area the rendering will appear smoothly but suddenly as the camera pans around a bend in the path the framerate will drop down to unplayable rates. In the end one of the best ways to judge framerate stability is by eye yourself, of course in this situation we rely on graphs and figures to present the situation to the reader.

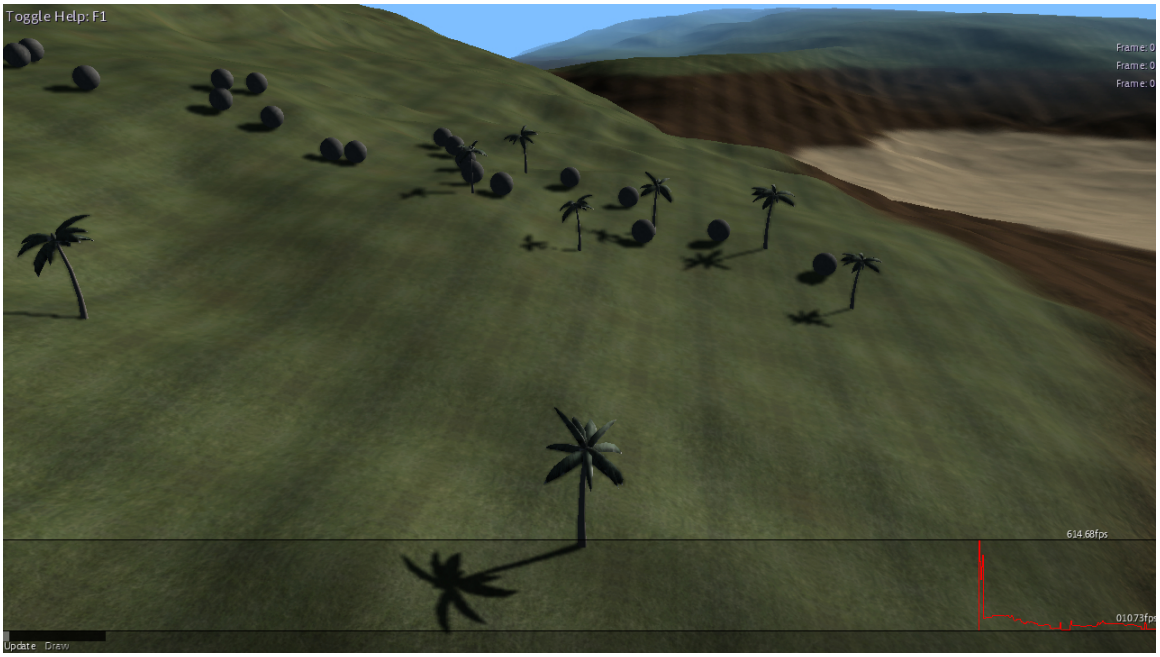

Figure 5.7: Rolling stones pass a cluster of palm trees.

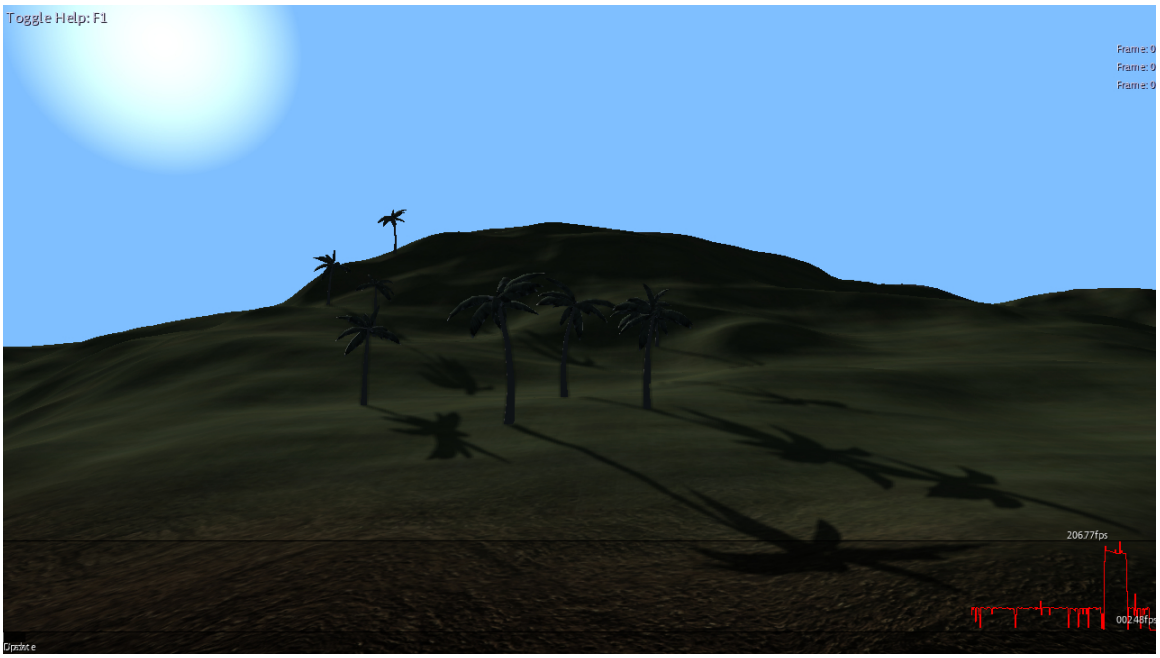

Figure 5.8: Trees cast long shadows in the dawn sunlight.

Chapter 6

Conclusions

6.1 Conclusions

The intention of this dissertation was to examine whether or not integrating ray-tracing elements into the rasterisation pipeline was feasible and beneficial. We have proven that it is indeed feasible to do so and even quite a simple matter when using a well encapsulated rendering pipeline like deferred lighting. However it cannot currently be shown to be beneficial to the final visual quality of the image, atleast not while maintaining an interactive framerate.

The results of the benchmarks performed clearly show that hardware is not powerful enough at the moment to perform ray-tracing in expansive dynamic game worlds. The framerates resulting from using this method are too unstable for use in a game engine today. In some areas with few objects the amount of time needed per frame for ray-tracing is not too excessive, however we just need to look at screenshots from any recent first-person shooter or other highly detailed game to see that there are rarely areas left with few objects in them anymore. Indeed most games like these are based in urban areas where the object density is far higher then the scenes we used for testing. As stated before the hierarchical uniform grid used could theoretically perform well in an urban scene but we did not have time to test this in our benchmarks. It is unlikely however that this would have offset the performance loss caused by so many objects in each node. All that would be needed to cripple performance would be a few hundred

thousand rays having to test against a node with dozens of highly detailed models. As our tests showed when nodes start to fill with complex models framerates will drop to very low levels, far below what the average gamer would consider acceptable.

This is just another area that should be tested in the future, along with other possibilities such as combining both shadow mapping and ray-traced shadows. One idea would be to use shadow buffer maps for the suns shadows, an area where ray-tracing performs quite slowly due to the fact that every pixel must be tested, and only use ray-tracing for things like local point lights where not as many rays would normally need to be shot and those that do wouldn't need to traverse through too many nodes. This sort of combined solution is one hybrid compromise that we might see in games in the near future. I am sure that there will be many more ingenious solutions developed over the coming years that combine the benefits of both ray-tracing and rasterisation to produce high image quality without affecting performance too badly.

We have looked at some of the various optimisations that can be used in order to increase performance and discussed the benefits and drawbacks of these. We have also examined which scene acceleration methods best suit a large, interactive, dynamic world and which are quick to translate to a format easily transferable to the GPU. We also discussed the benefits of performing ray-tracing on the SIMD architecture of the GPU and some of the performance pitfalls that can result.

Overall what should hopefully be taken from this paper is the idea that this is a feasible way forward for increasing visual definition in computer games in the future. Hardware will only continue to increase in performance and the software and programmable APIs that go with it will continue to improve as well. While it may not be of particular benefit on today's best GPUs in two years time these will be outdated and slow. The world of computer hardware, and in recent years particularly the graphics acceleration side of it, is a never-ending story of improvement upon improvement. It is only a matter of time, research and hard work until we see ray-tracing entering the world of the computer game.

6.2 Future Work

It is likely that ray-traced graphics, when they do start to be used in games, will be used in small amounts here and there in the rendering pipeline. Reflections, refractions and shadows are just some of the areas that will benefit from having GPU-accelerated ray-tracing. Physics could also benefit, we already see dynamic surfaces such as cloth and water being calculated on the GPU but these simulations usually do not house the entire scene's geometry on the GPU, instead just calculating interactions with some specific things like bullets and wind. If the scene already resides on the GPU in an easily accessible format then such simulations will be able to be expanded greatly.

Even on today's hardware it is very likely that a much more efficient ray-tracer could be developed given the appropriate time, experience and programming architecture. One of the major problems with this solution is the limitations imposed by directX 9. It was just never designed to cope with such dynamic branching and this is clearly evidenced by the compiling issues. A ray-tracer that took advantage of directX 11's more open nature or even the openCL programming framework would very likely perform much faster then our design. This is an area of research that strongly deserves more examination by people with more time and a better understanding of programming on the GPU.

The different scene acceleration methods available should also be researched further, object based bounding volume hierarchies might actually not be as slow as initially thought when the acceleration structure for this application was being decided upon and it might prove quicker during the traversal phase then our current hierarchical uniform grid. These are issues which, given more time, would have been tested against eachother in order to determine which best suited our interactive scene. As the popularity of ray-tracing in interactive games increases hopefully we will see such possibilities be researched both in the academic world and by game studios in their own engines for a more realistic sense of what ray-tracing is capable of when all other aspects of the game engine are also taken into account.

6.3 Final Thoughts

In the end we can only be happy with what we have accomplished, given more time more could have been done but it is only a matter of time before the issue of hybrid rendering becomes a mainstream concern for game developers. At that stage the issues brought up in this paper will be examined by dozens of teams across the world and we can be sure that the most efficient solutions to these problems will be found and the most made of what ray-tracing offers us.

Ray-tracing will eventually make it's way into game engines and when it does the results should be very impressive. The game industry has shown time and again their ability to make stunning visuals from resources that at first glance would appear nowhere near powerful enough. All we have to do is look at what has been achieved through tireless optimisation in the world of gaming consoles to see what can be made possible with limited hardware. If this same amount of optimisation is put into ray-tracing we will see some incredible feats performed by the talented development teams across the world. This could be the start of one of the biggest changes in computer game rendering since hardware acceleration.

Appendix A

Appendix

A.1 The XNA Framework

Microsoft's XNA is an API and a set of tools designed to help game developers avoid the need to write "repetitive boilerplate code" [40]. At its core XNA is a library of objects and methods commonly used in the development of both 2D and 3D games. Traditionally game developers would have to write their own classes for objects such as quaternions and matrices as well as all the computations that go with them. XNA does this for you and allows the developer to concentrate more on writing good gameplay and less on getting these details perfectly correct.

XNA also simplifies the setting up of the graphics device and communication between the CPU and GPU. This allows more people to get into game development without needing a massive knowledge of the intricacies of such things. In a world where more and more possibilities exist for amateur developers to publish their products it is necessary of course to give them the tools with which to make these games without the need to have completed an academic degree in the subject.

The primary principle behind Microsoft's development of XNA is code repetition. They recognised the fact that people throughout the world were writing what was essentially identical code for their mathematical objects and formulas. They took these pieces of code and placed them into one easy to use library which since its release in March 2006

has become one of the most popular tools of amateur game developers.

This test uses XNA for two main reasons. The first being what is mentioned above, the ease with which to set the test up which allows far more time to be given to the main problem at hand, that of combining rasterisation and ray-tracing efficiently on the GPU. The second, and perhaps more important is the fact that amateur developers will be able to take the result of these tests and implement the solution directly into their own projects and hopefully improve upon it. Many of the great advances in computing have started out in someone's bedroom and these tests, and XNA itself embrace that ideal.

A.2 DirectX 9

DirectX 9 was first released in December 2002 and since then has proved to be the longest running version of the API with updates still common til this day, the latest at the time of writing being June 2010. One of the primary reasons for this was the controversial decision by Microsoft to limit future versions of directX, namely versions 10 and 11 to their new operating systems Vista and Windows 7. Another reason is the limitations of the current generation of consoles to directX 9 instructions, particularly Microsoft's own Xbox 360.

DirectX 9 was also the first version of the API to feature true programmable shaders with the High Level Shading Language(HLSL), it was a massive improvement over the fixed function pipeline of earlier years and was to lead to the development of stunning new effects in games. It is a testament to its design that it is still in use today in the vast majority of games and also that it can cope with just about any programming challenge thrown at it, including ray-tracing on the GPU.

That being said however it is not without its flaws, design constraints which seemed generous in early years now lead to problems with more complex shaders, in particular GPGPU shaders. A register limit of 224 is imposed in the latest version of directX 9.0c with a dynamic branching limit of 24. These have lead to numerous issues throughout the development of this ray-tracing application where dynamic branching is critical in

breaking loops, discarding the ray upon intersections, deciding whether to test against the object meshes after a bounding sphere test and a myriad of other cases. It has been a constant struggle with the programming language itself to get past these problems. In particular the compile time of the primary ray-tracing shader with all functionality enabled grew to be so long that for a while it was not possible to tell whether it would even be possible to compile it in directX 9.

A.3 SlimDX and DirectX 11

The issues were so great that with just weeks left in the development of the application SlimDX was considered as an alternative framework to XNA. SlimDX is similar in some ways to XNA, it is written in C-sharp and it is a wrapper for DirectX. One of the benefits of SlimDX is that it allows development for any version of directX from 9.0 onwards. This would mean that the shader could be written and compiled for directX 11 which has a much higher register count and essentially infinite amount of dynamic branching available. Trying to compile the shader in directX 11 proved to solve the problems of long compile times, the shader was compiled in mere seconds instead of the 45 minutes or so of compile time for directX 9.

Due to the limited time left available for work on the implementation it was however decided to stay with XNA. The engine behind the application is quite large with a lot of the key functionality relying on utilities provided by XNA. If the framework was changed mid-build all this functionality would have to be either written or alternative utilities in the SlimDX library found. It was easier, although awkward, to write the ray-tracing code piecemeal in XNA, building and testing one section at a time which lead to relatively low compile times of just a few minutes. Then, every few days a full shader build was performed to ensure that it was all still working well together. This method of building in parts ensured not only that programming could proceed at greater speed but that each section was well encapsulated and could be turned on or off at will. Such improvements allowed for the introduction of primitive types like spheres for example where when an object is tested it relies purely on the original bounding sphere test and does not move on to the mesh intersection tests which would prove highly time consuming for a polygon heavy sphere.

A.4 Application Controls

Figure A.1 shows the keyboard control layout for the application. In addition to this the standard WASD keys and mouse are use to control the cameras movement. The left mouse button selects objects and the middle mouse button will clone a new palm tree object.

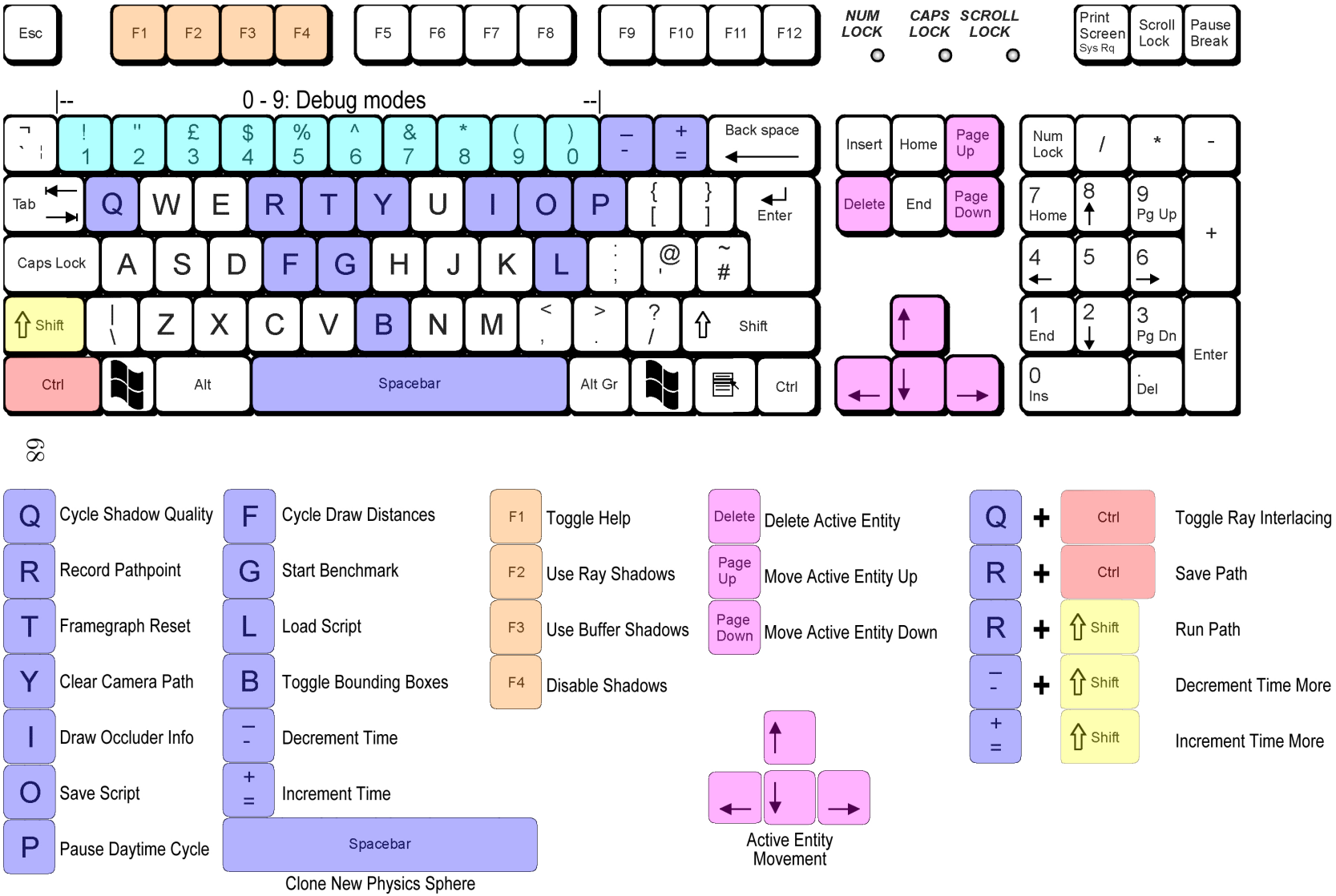

Figure A.1: Key Controls for the Application

Bibliography

[1] http://en.wikipedia.org/wiki/Kd_tree, "Wikipedia entry on k-d trees, august 2010."

[2] http://www.amd.com/us/products/desktop/graphics/ati-radeon-hd-5000/hd-5970/Pages/ati-radeon-hd 5970-specifications.aspx, "Ati radeon 5970 spec sheet, march 2010."

[3] http://www.cryengine2.com/, "Cryteks cryengine 2, june 2010."

[4] http://www.unrealtechnology.com/technology.php, "id softwares unreal engine 3, march 2010."

[5] http://developer.nvidia.com/object/Cube_Mapping_Paper.html, "nvidia paper on cube reflection mapping, july 2010."

[6] A. Appel, "Some techniques for shading machine renderings of solids," in *AFIPS '68 (Spring): Proceedings of the April 30–May 2, 1968, spring joint computer conference*, (New York, NY, USA), pp. 37–45, ACM, 1968.

[7] http://developer.nvidia.com/object/optix home.html, "nvidia optix ray-tracing engine, july 2010."

[8] T. Aila and S. Laine, "Understanding the efficiency of ray traversal on gpus," in *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, (New York, NY, USA), pp. 145–149, ACM, 2009.

[9] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek, "Realtime ray tracing on gpu with bvh-based packet traversal," in *RT '07: Proceedings of the 2007 IEEE*

Symposium on Interactive Ray Tracing, (Washington, DC, USA), pp. 113–118, IEEE Computer Society, 2007.

[10] D. R. Horn, J. Sugerman, M. Houston, and P. Hanrahan, "Interactive k-d tree gpu raytracing," in *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, (New York, NY, USA), pp. 167–174, ACM, 2007.

[11] http://gpurt.sourceforge.net/DA07_0405_Ray_Tracing_on_GPU 1.0.5.pdf, "Ray tracing on gpu, martin christen, march 2010."

[12] http://www.pcper.com/article.php?aid=532&type=expert&pid=1, "John carmack on id tech 6, ray tracing, consoles, physics and more, may 2010."

[13] http://www.pcper.com/article.php?aid=546&type=expert, "Crytek's cevat yerli speaks on rasterization and ray tracing, may 2010."

[14] I. Wald and P. Slusallek, "State of the art in interactive ray tracing," 2001.

[15] http://www.win.tue.nl/ hermanh/stack/bvh.pdf, "Introduction to bounding volume hierarchies, august 2010."

[16] http://tog.acm.org/resources/RTNews/html/rtnews1b.html, "Spline surface rendering, and what's wrong with octrees., august 2010."

[17] I. Wald, "On fast construction of sah-based bounding volume hierarchies," in *RT '07: Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, (Washington, DC, USA), pp. 33–40, IEEE Computer Society, 2007.

[18] M. Shevtsov, A. Soupikov, and E. Kapustin, "Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes."

[19] M. Eitz and G. Lixu, "Hierarchical spatial hashing for real-time collision detection," in *SMI '07: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007*, (Washington, DC, USA), pp. 61–70, IEEE Computer Society, 2007.

[20] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross, "Optimized spatial hashing for collision detection of deformable objects," in ., pp. 47–54, 2003.

[21] http://en.wikipedia.org/wiki/Differential_Analyzer, "Wikipedia entry on differential analysers, august 2010."

[22] A. Pollen, *The Great Gunnery Scandal The Mystery of Jutland*. Collins, 1980.

[23] http://en.wikipedia.org/wiki/Digital_Differential_Analyzer, "Wikipedia entry on digital differential analysers, august 2010."

[24] http://en.wikipedia.org/wiki/Digital_Differential_Analyzer_(graphics_algorithm), "Wikipedia entry on graphical digital differential analysers, august 2010."

[25] http://www.qwrt.de/, "Quake wars: Ray traced, april 2010."

[26] R. L. Cook, L. Carpenter, and E. Catmull, "The reyes image rendering architecture," *SIGGRAPH Comput. Graph.*, vol. 21, no. 4, pp. 95–102, 1987.

[27] P. Christensen, "Ray tracing for the movie "car"," *Symposium on Interactive Ray Tracing*, vol. 0, p. ix, 2006.

[28] http://www.bth.se/fou/cuppsats.nsf/all/b27da4d4733a0234c12575c90044f834/$file/Henrik_Poulsen_Hybrid_Ray_Tracer.pdf, "Potential of gpu based hybrid ray tracing for real time games by henrik poulsen, august 2010."

[29] http://en.wikipedia.org/wiki/S.T.A.L.K.E.R.:_Shadow_of_Chernobyl, "Wikipedia entry for stalker, june 2010."

[30] http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter09.html, "Gpu gems2 - chapter 9. deferred shading in s.t.a.l.k.e.r., july 2010."

[31] http://www.incrysis.com/index.php?option=com_content&task=view&id=797, "Deferred lighting in cryengine 3, july 2010."

[32] http://developer.amd.com/gpu_assets/S2008 Filion-McNaughton-StarCraftII.pdf, "Star craft 2: Effects and techniques, august 2010."

[33] http://http.download.nvidia.com/developer/presentations/2004/6800_Leagues/6800_Leagues_Deferred_Shading.pdf, "6800 leagues under the sea: nvidia deferred shading, august 2010."

[34] http://www.nvidia.com/object/product_geforce_gtx_470_us.html, "Official nvidia gtx470 specifications, august 2010."

[35] http://jiglibx.codeplex.com/, "Jiglibx physics engine, august 2010."

[36] http://www.rowlhouse.co.uk/jiglib/index.html, "Jiglib physics engine, august 2010."

[37] http://www.mafia2game.com/, "Mafia ii, august 2010."

[38] http://www.nvidia.com/object/physx_new.html, "nvidias physx engine, july 2010."

[39] http://en.wikipedia.org/wiki/ASCI_Red, "Wikipedia entry on intel's asci red supercomputer, september 2010."

[40] https://www.microsoft.com/presspass/press/2004/mar04/03
24xnalaunchpr.mspx, "Microsoft: Next generation of games starts with xna, july 2010."