Comparing the Traversal of Acceleration Data Structures for Real-time Ray Tracing

by

Sean Legg, B.Sc. Computer Science

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 2013

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.

Sean Legg

August 27, 2013

Permission to Lend and/or Copy

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

____
Sean Legg

August 27, 2013

Acknowledgments

I would like to thank my supervisor Michael Manzke for his advice and guidance throughout this dissertation, I would also like to thank Michael John Doyle for his advice also. Lastly I would like to thank anyone who gave me support during the year and on my dissertation, including the members of the IET course, friends and family.

Sean Legg

University of Dublin, Trinity College
September 2013

Comparing the Traversal of Acceleration Data Structures for Real-time Ray Tracing

Sean Legg
University of Dublin, Trinity College, 2013

Supervisor: Dr. Michael Manzke

The purpose of this research is to compare the performance of the traversal of acceleration data structures looking at the characteristics that the acceleration data structure exhibits. In particular I will be looking at the performance of bounding volume hierarchies (BVH) and K-d (K-dimensional) trees. A number of experiments will be carried out on these acceleration data structures at a low-level looking at phonenomena such as cache performance and branch divergence.

Contents

Acknowledgments **iv**

Abstract **v**

List of Tables **ix**

List of Figures **xi**

Chapter 1 Introduction **1**

1.1 Motivation 1

1.2 Report Roadmap 2

Chapter 2 State of the Art **3**

2.1 GPGPU 4

2.2 Bounding Volumes 4

2.2.1 Bounding Sphere 5

2.2.2 AABB 6

2.2.3 OBB 6

2.3 Acceleration Data Structures 6

2.3.1 Bounding Volume Hierarchies 7

2.3.2 K-d trees 8

2.3.3 Uniform Grids 9

2.3.4 Hierarchical Grids 10

2.3.5 BSP Trees 11

2.4 Specialised Hardware 12

2.5 Non-Real-Time Applications 12

2.6 Real-time Applications 13

Chapter 3 Method **16**

3.1 Models 16

3.1.1 The Stanford Models 16

3.1.2 BART 17

3.1.3 MGF 19

3.2 Traversal Time 19

3.3 Cache Performance 21

3.3.1 16kb L1 with 48kb Shared Memory 22

3.3.2 48kb L1 with 16kb Shared Memory 22

3.3.3 L1 Cache Disabled 22

3.4 Branch Divergence 22

3.5 Instruction Statistics 23

3.6 Dynamic Scene 24

Chapter 4 Test Setup **26**

4.1 Hardware 26

4.2 Tools 27

4.2.1 Nvprof 27

4.2.2 NSight 27

4.3 Fermi Architecture 27

4.3.1 Streaming Multiprocessors 28

4.3.2 Configurable L1 / Shared Memory Cache 28

4.4 Implementation 30

4.4.1 K-d Tree 30

4.4.2 BVH 33

Chapter 5 The Results **36**

5.1 Full Render 36

5.1.1 Stanford 36

5.1.2 BART 39

5.1.3 MGF 43

5.2 Traversal Time Per Pixel 47

5.2.1 Stanford 47
5.2.2 BART 51
5.2.3 MGF 55
5.2.4 Dynamic Scene Traversal 59
5.3 Cache Performance 59
5.3.1 Stanford 60
5.3.2 BART 60
5.3.3 MGF 60
5.3.4 Dynamic Scene 69
5.4 Branch Divergence 70
5.4.1 Stanford 70
5.4.2 MGF 71
5.4.3 BART 72
5.5 Instruction Statistics 74
5.5.1 Stanford 74
5.5.2 BART 74
5.5.3 MGF 78

Chapter 6 Conclusions & Future Work **85**
6.1 Conclusion 85
6.2 Future Work 86

Appendix A List of nvprof Metrics **88**

Bibliography **96**

List of Tables

3.1 Stanford Models - Vertex & Triangle Count [33] 17

4.1 Hardware Setup 26
4.2 Fermi Specifications 29

5.1 Stanford Primary Ray Full Render 37
5.2 Stanford Primary & Shadow Rays Full Render 38
5.3 BART Robots Traversal Time K-d tree and BVH 40
5.4 BART Museum Traversal Time K-d tree and BVH 41
5.5 BART Kitchen Traversal Time K-d tree and BVH 42
5.6 MGF Theatre Traversal Time K-d tree and BVH 44
5.7 MGF Conference Traversal Time K-d tree and BVH 45
5.8 MGF Office Traversal Time K-d tree and BVH 46
5.9 Stanford Bunny Average Traversal Times (Per Pixel) 48
5.10 Stanford Dragon Average Traversal Times (Per Pixel) 49
5.11 Stanford Buddha Average Traversal Times (Per Pixel) 50
5.12 BART Kitchen Average Traversal Times (Per Pixel) 52
5.13 BART Museum Average Traversal Times (Per Pixel) 53
5.14 BART Robots Average Traversal Times (Per Pixel) 54
5.15 MGF Conference Average Traversal Times (Per Pixel) 56
5.16 MGF Office Average Traversal Times (Per Pixel) 57
5.17 MGF Theatre Average Traversal Times (Per Pixel) 58
5.18 Dynamic Scene Primary Ray Average Traversal Times 59
5.19 Dynamic Scene Primary & Shadow Rays Average Traversal Times (Per-Pixel) 59

5.20 Stanford K-d Primary Ray Cache Hit Rates 61
5.21 Stanford K-d Primary & Shadow Ray Cache Hit Rates 62
5.22 BART Robots Cache Hit Rates 63
5.23 BART Kitchen Cache Hit Rates 64
5.24 BART Museum Cache Hit Rates 65
5.25 MGF Conference Cache Hit Rates 66
5.26 MGF Office Cache Hit Rates 67
5.27 MGF Theatre Cache Hit Rates 68
5.28 Dynamic Scene Cache Hit Rates 69
5.29 Stanford Bunny Branch Divergence 70
5.30 Stanford Dragon Branch Divergence 70
5.31 Stanford Buddha Branch Divergence 71
5.32 MGF Theatre Branch Divergence 71
5.33 MGF Conference Branch Divergence 71
5.34 MGF Office Branch Divergence 72
5.35 BART Robots Branch Divergence 72
5.36 BART Kitchen Branch Divergence 73
5.37 BART Museum Branch Divergence 73
5.38 Instructions Executed Stanford Bunny 75
5.39 Instructions Executed Stanford Dragon 76
5.40 Instructions Executed Stanford Buddha 77
5.41 Instructions Executed BART Robots 79
5.42 Instructions Executed BART Museum 80
5.43 Instructions Executed BART Kitchen 81
5.44 Instructions Executed MGF Conference 82
5.45 Instructions Executed MGF Office 83
5.46 Instructions Executed MGF Theatre 84

A.1 nvprof events 95

List of Figures

2.1 Bounding Volumes [5] 5
2.2 Bounding Sphere Collision Detection 5
2.3 AABB Collision Detection [5] 6
2.4 Example of a bounding volume hierarchy [27] 7
2.5 Example of K-d tree spatial subdivision [22] 8
2.6 Example of a Uniform Grid [26] 9
2.7 DDA Traversal [23] 10
2.8 Example of a Hierarchical Grid [5] 11
2.9 Example of a BSP tree [5] 12
2.10 The movie 'Cars' by Pixar using ray-traced reflections [3] 13
2.11 NVidia OptiX Cook Demo 14

3.1 Stanford Models Render [33] 17
3.2 BART Scenes [21] 18
3.3 MGF Scenes [17] 20
3.4 Traversal Time Experiment 21
3.5 CUDA cache configuration options 21
3.6 CUDA Cache Configuration 22
3.7 CUDA Disable L1 Cache 22
3.8 Branch Divergence nvprof 23
3.9 Branch Divergence [11] 24
3.10 Divergent Branch Percentage [6] 24
3.11 Instructions Executed with nvprof 24
3.12 Dynamic Scene Motion 25

4.1 NVidia Nsight Profiler 28
4.2 NVidia Fermi Architecture [24] 30
4.3 Streaming Multiprocessor Overview [24] 31
4.4 Typical K-d Traversal Psuedocode [12] 32
4.5 kd-restart Algorithm 33
4.6 GPU BVH Traversal Psuedocode [12] 35

5.1 Dragon Primary Rays Instructions Executed 74

Chapter 1

Introduction

Since the early days of computer graphics we have used rasterisation based techniques for rendering which rely on a number of approximations to achieve certain visual effects. Ray tracing offers an alternative approach in which it achieves better photorealism and removes the need for approximation techniques. We can also combine both rasterisation and ray-tracing techniques to get the best of both.

One of the major drawbacks with ray tracing is the fact that it is computationally very expensive and thus has not been used in any commercial games but has been used in movies as the scenes can be rendered offline.

1.1 Motivation

The motivation of this report is to compare both K-d trees and bounding volume hierarchies (BVH) at a low level to see what characteristics they exhibit as this is previously unknown.

One paper of interest is the paper, "Ray Tracing on a GPU with CUDA Comparative Study of Three Algorithms" [38] which does a excellent comparison of these data structures, I intend to extend this research by looking at low level characteristics of the acceleration data structures such as the cache performance and branching. I will also be looking at the traversal times for the data structures but not for the entire CUDA

kernal which includes other calculations such as shading.

1.2 Report Roadmap

This report is structured as follows:

Chapter 2 “State of the Art” covers the state-of-the-art techniques used in ray-tracing, this chapter mainly looks at the acceleration data structures commonly used for ray-tracing.

Chapter 3 “Methods” will discuss the experiments in detail and how I plan to carry them out.

Chapter 4 “Test Setup” covers items such as the hardware setup used and the code base used for the experiments.

Chapter 5 “The Results” will present and discuss the results achieved from the experiments.

Chapter 6 “Conclusion” is focused on possible future work and what this project achieved.

Chapter 2

State of the Art

The most basic ray-tracing algorithm shoots rays into a scene through a virtual camera and checks for intersections with triangles within the scene. Computationally this is very expensive If you are running at a large resolution such as 1920x1080 then you have to check intersections with every triangle in the scene with over 2 million rays. If we also consider the complexity of modern game engines which can contain millions of triangles in a single scene, this means that even in a scene with 1 million triangles we could have roughly 2 million x 1 million iterations for the ray tracer which is two trillion checks per update and is not feasible for real-time ray tracing.

To get past this limitation we employ the use of acceleration data structures which lower the number of checks we need to perform for a single ray by splitting the scene up. Each of these acceleration data structures have their own advantages and disadvantages. The two types of acceleration data structures object and location based, object based approaches group objects together based on how close they are to each other whereas location based approaches look at the objects location in the scene to decide whether or not to check for a collision.

Another technique we use is to make use of the GPU (graphics processing unit) instead of the CPU (central processing unit) to perform ray tracing, Intel's latest CPU give us around 100 GFLOPs compared to the 4.5 TFLOPS of the NVidia GTX Titan. [29]

2.1 GPGPU

In recent years ray-tracing has moved the CPU to the GPU which exploits low-level parallelism. When performing ray-tracing we gain a several fold performance improvement as we can generate a thread on the GPU for a ray that shoots through the virtual camera which means that each ray is computed in parallel.

The use of GPUGPU in general can have a huge performance increase over CPUs, in the case of Massachusetts General Hospital who used CUDA to perform Monte Carlo simulation on the GPU they found that they had a huge 300x performance increase over the use of a CPU though in most cases the performance increase is not this high. In the paper "Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU" [20] from a team at Intel they found that the GPU had an average 2.5 times performance increase over the CPU, NVidia later responded to this paper on their blog giving real-world examples that had 100 to 300 times speedups over the CPU.

The paper "Stackless KD-Tree Traversal for High Performance GPU Ray Tracing"[32] looks at ray-tracing on the GPU and the CPU, they found that there was a tenfold performance increase when using the GPU compared to the CPU when comparing primary rays only with packet ray traversal. It is also worth noting that this paper was from 2007 where they used an NVidia 8800GTX which has around 500 GFLOPs throughput compared to the 4.5 TFLOPs of the NVidia GTX Titan which is a huge increase compared to the latest CPUs, another thing to note is that GPUs are scalable in the case of NVidia we can use their SLI technology to run multiple graphic cards in a single system with a maximum of four GPUs and in the case of AMD we can use their crossfire technology.

2.2 Bounding Volumes

To prevent performing intersection tests with every polygon of an object bounding volumes are typically used, bounding volumes are simple primitive objects that are used to encapsulate a complex game object. There are a number of different bounding

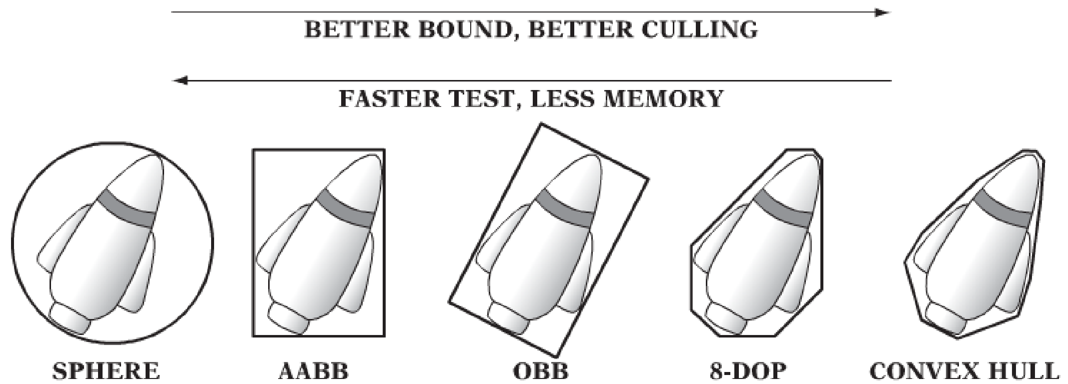


Figure 2.1: Bounding Volumes [5]

volumes available, the more complex the bounding volume is to fit the better the culling of objects but this comes at the cost of more complex collision detection as shown in figure 2.1. [5]

2.2.1 Bounding Sphere

A common bounding volume is the bounding sphere which wraps a game object within a sphere, this is the simplest bounding volume but typically does not provide a tight fit around objects. To perform collision checks between the spheres we check if the distance between the center of two spheres is less then their combined radii as shown in figure 2.2.

```
bool CheckSphereSphere(BoundingSphere* a, BoundingSphere* b)
{
    glm::vec3 aPos = a->GetPosition();
    glm::vec3 bPos = b->GetPosition();

    if (glm::distance(aPos, bPos) < a->GetRadius() + b->GetRadius())
        return true;
    return false;
}
```

Figure 2.2: Bounding Sphere Collision Detection

2.2.2 AABB

Another type of bounding volume is the axis-aligned bounding box (AABB) which is essentially a box that encapsulates the object, while this provides a better fit then bounding spheres the collision detection is slightly more complex. We can check for collisions between two AABBs by checking the min and max positions for each axis as shown in figure 2.3.

```
int TestAABBAABB(AABB a, AABB b)
{
    if ((t = aMin->x - bMin->x) > bDiameter->x || -t > aDiameter->x) return false;
    if ((t = aMin->y - bMin->y) > bDiameter->y || -t > aDiameter->y) return false;
    if ((t = aMin->z - bMin->z) > bDiameter->z || -t > aDiameter->z) return false;

    return true;
}
```

Figure 2.3: AABB Collision Detection [5]

2.2.3 OBB

Oriented bounding boxes (OBB) are similar to an AABB except that they also have an orientation, this type of bounding volume typically fits an object better than an AABB. The collision check between OBBs is more complex and requires 15 tests to determine intersection. [5]

2.3 Acceleration Data Structures

In order to achieve high performance real-time ray tracing we make use of acceleration data structures which involves breaking down the scene into manageable chunks with the goal of avoiding collision checks with every triangle or object in the scene. There are a number of different acceleration structures used and also some hybrid structures which make use of multiple acceleration data structures. [5]

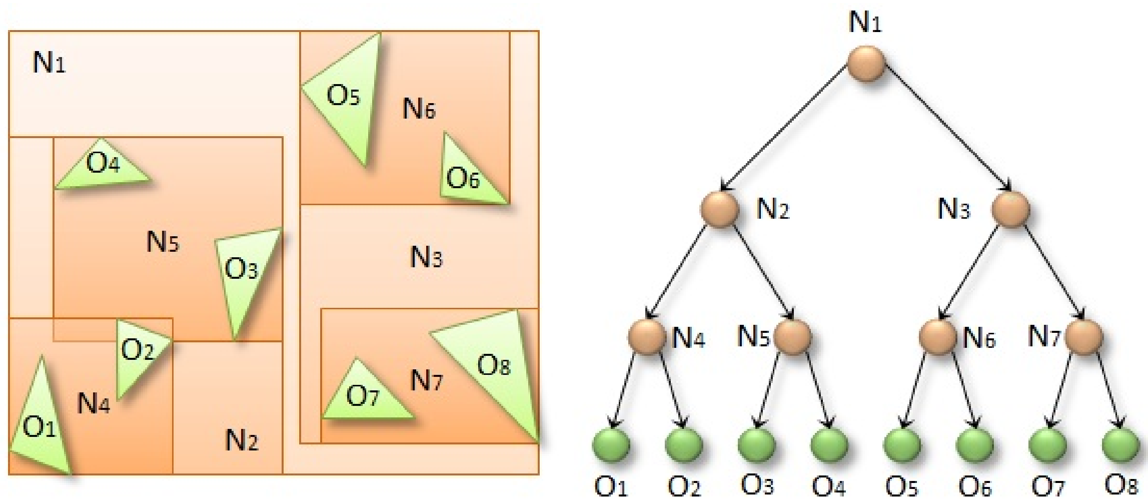


Figure 2.4: Example of a bounding volume hierarchy [27]

2.3.1 Bounding Volume Hierarchies

One of these acceleration data structures is bounding volume hierarchies (BVH) which is a tree structure made up of bounding volumes such as a sphere or an axis-aligned bounding box (AABB), on the leaf nodes of the tree we find the geometric objects themselves. The use of a BVH results in a significant performance improvement, figure 2.4 shows an example of a BVH structure where N1 is the root node and O1 to O8 are the leaf nodes.

When checking for collisions between a ray and objects we start at the root node of the tree and see if the ray intersects the bounding volume, if this bounding volume is intersected then we move to the next node in the tree which is dependent on which heuristic we use to traverse the tree such as depth-first search which will iterate deeper into the tree or breadth-first search which iterates horizontally over the tree before proceeding deeper into the tree. We gain a performance improvement by eliminating branches from the tree for example in 2.4 if the ray does not intersect with the bounding volume at N3 then we can cut off that branch completely which removes the need to check any of the child nodes of that branch whereas without the BVH we would have to check every object in the scene. [27]

One issue with the use of a BVH is we need to rebuild the tree when objects move which can be costly, the paper "Fast BVH Construction on GPUs" [19] looks at this problem and found that when using a complex scene with 1.5 million triangles it took 66ms to build the tree which is quite a long time considering that we need to render in 16.67ms to achieve 60 frames per second or 33.34ms to render at 30 frames a second (Which should be an absolute minimum frame rate) and that the 66ms is only the cost to build the tree and not to render the scene.

When we build the BVH tree one important heuristic is the surface area heuristic (SAH) which is used to control primitive splitting during the construction of the tree. The way that the tree is split will also have a direct effect on the performance of the tree traversal when it comes to ray-tracing. [35]

2.3.2 K-d trees

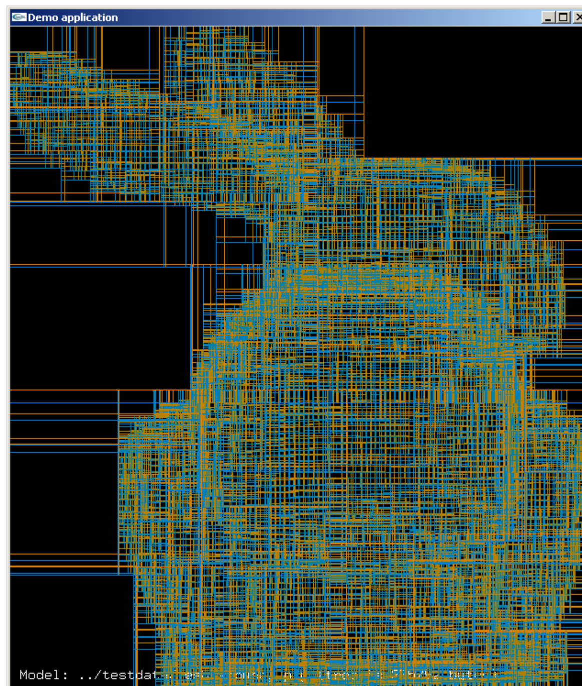


Figure 2.5: Example of K-d tree spatial subdivision [22]

K-d trees or K-dimensional trees are a generalisation of octrees and quadtrees that

are used to partition space. They work by partitioning space one dimension at a time where k represents the number of dimensions. The axis are typically split in a cyclic manner in which the x-axis is split first followed by the y-axis and z-axis this is then continued by splitting the x-axis again and so forth. Figure 2.5 shows an example of a three dimensional k-d tree where the scene has been subdivided into a number of smaller regions. [5]

When it comes to collision detection k-d trees can be used in situations where either octrees or quadtrees are used. They can also be used to check the point location, given a point the region in which it resides can be found. Nearest neighbour searches can be performed with a k-d tree in which it will find the point from a set a points the query point is closest to. [12] [5]

2.3.3 Uniform Grids

Figure 2.6: Example of a Uniform Grid [26]

A uniform grid is a simple effective spacial subdivision scheme which is used to partition space with a regular grid. The grid is divided up into a number of cells each of

equal size, each object in the scene is associated with the cell that it overlaps, figure 2.6 shows an example of a uniform grid. The only objects that can be colliding are those which overlap a common grid cell in which case more in-depth collision checks can be used.

The performance for uniform grids is directly correlated to the size of the grid cells, a balance must be found where the cells are not too small or large and also where there are not a large amount of objects occupying the same grid cell.

In order to traverse a uniform grid we need to use a Digital Differential Analyser (DDA) algorithm which is used for linear interpolation of variables between two points, the start and end point. Figure 2.7 shows an example of a 3D-DDA. [23]

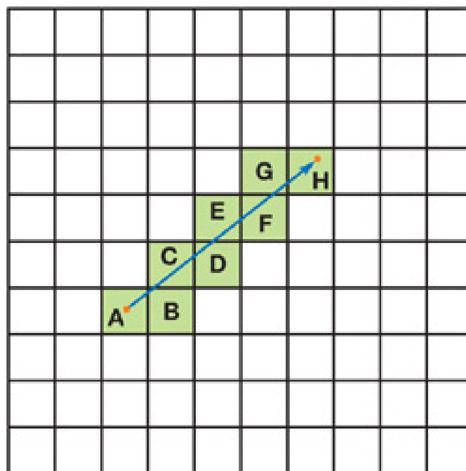


Figure 2.7: DDA Traversal [23]

2.3.4 Hierarchical Grids

A hierarchical grid attempts to solve the problem with uniform grids where it is difficult to deal with objects that vary greatly in size. With the use of uniform grids objects may span multiple grid cells leading to moving objects becoming expensive, hierarchical grids are a good alternative to uniform grids as they are well suited to holding dynamically moving objects.

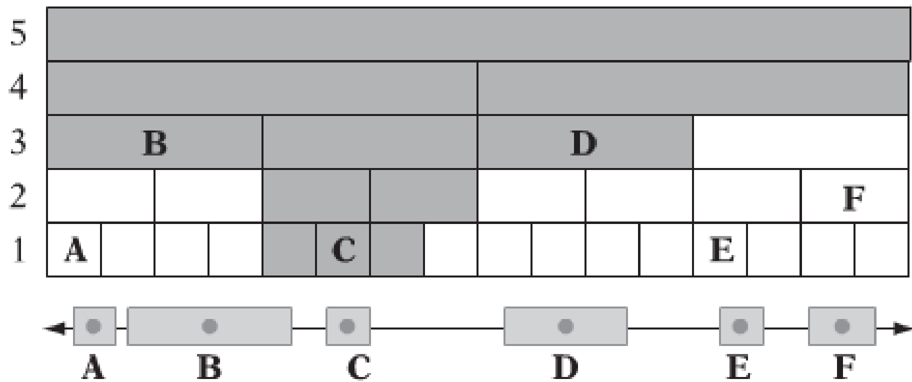


Figure 2.8: Example of a Hierarchical Grid [5]

Hierarchical grids work by traversing through all the hierarchical grid levels, at each level the object to be checked is compared to both the cells that its bounding volume overlaps and also the neighbouring cells that could have objects extending into the cells that are overlapped. Figure 2.8 shows the structure of a small one dimensional hierarchical grid where there are six objects A to F and the shaded area are where we must check for collisions for the object C. [5]

2.3.5 BSP Trees

Binary space-partitioning (BSP) trees are a structure used to partition space. This data structure works by recursivly diving space into subspaces based on planes or arbitrary position and orientation. [16]

Figure 2.9 shows the stages involved in the division of a square into four convex subspaces and also the corresponding tree structure. The first stage (a) shows the initial split, the second stage (b) shows the first second-level split and finally the last stage (c) shows the second second-level split. There are a number of different types of BSP trees.

The paper “Ray Tracing with the BSP Tree” implements a ray-tracer using BSP trees and they found that the performance was competitive with a ray-tracer using a BVH or K-d tree.

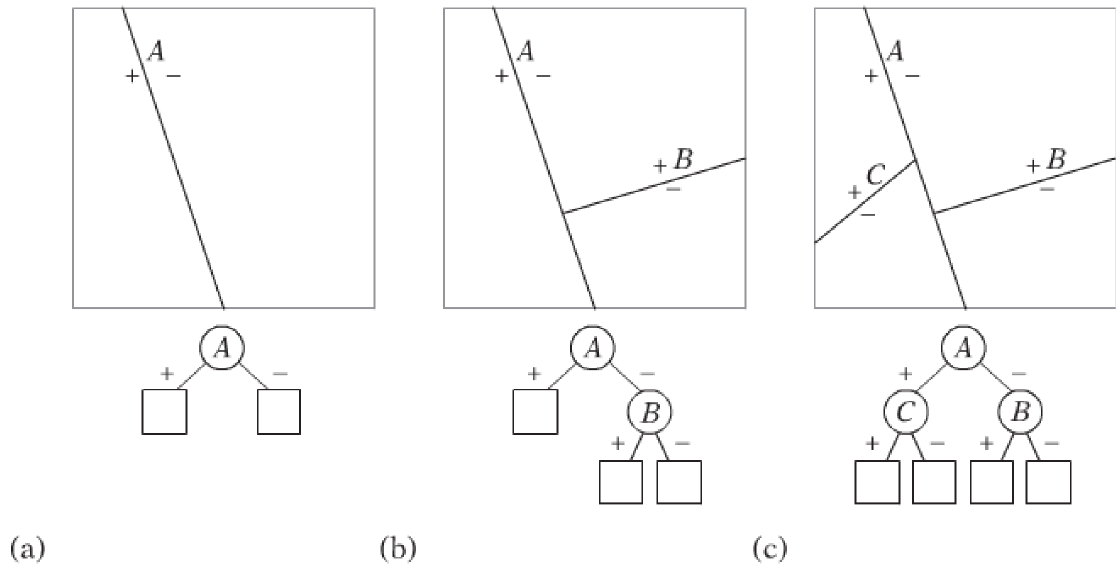


Figure 2.9: Example of a BSP tree [5]

2.4 Specialised Hardware

There is specialised hardware available that is designed purely for ray-tracing performance, an example of this is the R2500 by Imagination Technologies which supports up to 100 million incoherent rays per second or 50 million incoherent rays per second for the lower end model R2100. Unfortunately there is not a lot of information on these cards such as benchmarks comparing them to GPUs at the present time as these particular models have only recently been released. It should be noted that these cards are aimed at offline rendering and not real-time ray-tracing. [13]

2.5 Non-Real-Time Applications

The movie "Cars" by Pixar made use of ray tracing using their RenderMan renderer though they only used ray-tracing for the reflections found on the cars an example of the output can be seen in figure 2.10. In the paper "Ray Tracing for the Movie Cars" they describe the process of ray-tracing they used and the issues they encountered with the use of ray-tracing. They faced many issues when trying to add ray-tracing to the movie mainly due to the shear size of some of the scenes, one which they mention in the

Figure 2.10: The movie 'Cars' by Pixar using ray-traced reflections [3]

paper has a racing oval with 75,000 cars as spectators. One major challenge they faced was that the scene had to fit into memory otherwise the program would have to read from virtual-memory which would slow the algorithm down by orders of magnitude, they also had to look at ray-tracing objects that were outside of the current viewport. To get past a number of issues they were facing they made use of ray differentials, multiresolution geometry and texture caches in order to make the scene tracable. [3]

2.6 Real-time Applications

While there are no commercially available ray-traced games due to the large performance cost there has been research into ray-tracing some of the earlier Id tech games. One game of particular interest is "Quake Wars: Enemy Territory" which was modified to become "Quake Wars: Ray Traced" [14] by Intel, unfortunately the source code or demo was never released to the public. The implementation they used was based on the CPU which achieved between 15 and 35 frames depending on the system used. The successor to this project is "Wolfenstein: Ray Traced" [15] but again no source code

or demo has been released publicly.

Figure 2.11: NVidia OptiX Cook Demo

NVidia provide an SDK for ray-tracing called OptiX which makes use of a GPU with CUDA technology to run in real-time. Optix provides support for building and traversing a number of state-of-the-art acceleration data structures such as Kd-trees and bounding volume hierarchies with code that is optimized to run on the GPU in parallel. [31]

Along with Optix, NVidia provide a debugger and profiler called Nsight which is available an an extension to both visual studio and eclipse. Nsight provides support for running a number of experiments to profile an application, this includes a large amount of low-level performance metrics such as cache hits, instructions executed, branches taken, memory bandwidth along with many more which is useful for profiling an application to see what is causing any performance bottlenecks. [30]

In recent years there have been an increasing number of technical demos involving ray-tracing one notable demo was that from NVidia which showed ray-traced fluids with

destruction in real-time which was shown at GTC (GPU Technology Conference) 2013. This demo made use of OptiX 3.0 and ran off two quadro graphics cards in real-time.

Eventually it may be the case that ray-tracing will replace rasterisation due to a number of key factors, though this is unlikely to happen for quite some time. For one rasterisation based techniques rely on approximations to implement most rendering effects such as screen space ambient occlusion (SSAO). Ray-tracing scales better then rasterization based rendering as ray-tracing is logarithmic whereas rasterisation is linear. [1]

Chapter 3

Method

As part of this dissertation I will carry out a number of experiments on ray-tracing two acceleration data structures, the K-d tree and BVH. This section will describe in detail the experiments I plan to carry out and how I will go about them.

The tools used for this dissertation allow for a wide variety of experiments to be run, the full list of metrics supported by the command line tool nvprof can be found in the appendix table A.1. Out of these possible metrics I selected the ones that would show what characteristics these acceleration data structures exhibit such as their cache performance and traversal times.

3.1 Models

To test the performance of these acceleration data structures I will be using a number of different models. This section gives a brief overview of the models that are used, the Stanford, BART and MGF models.

3.1.1 The Stanford Models

The Stanford models are a set of high detail models which were created using a scanned and then processed to produce a single triangle mesh. The models I will specifically looking at from this set of models are the bunny, dragon and buddha models. [33]

Further details about these models can been seen in table 3.1 which shows the vertex and triangle counts for each of these models.

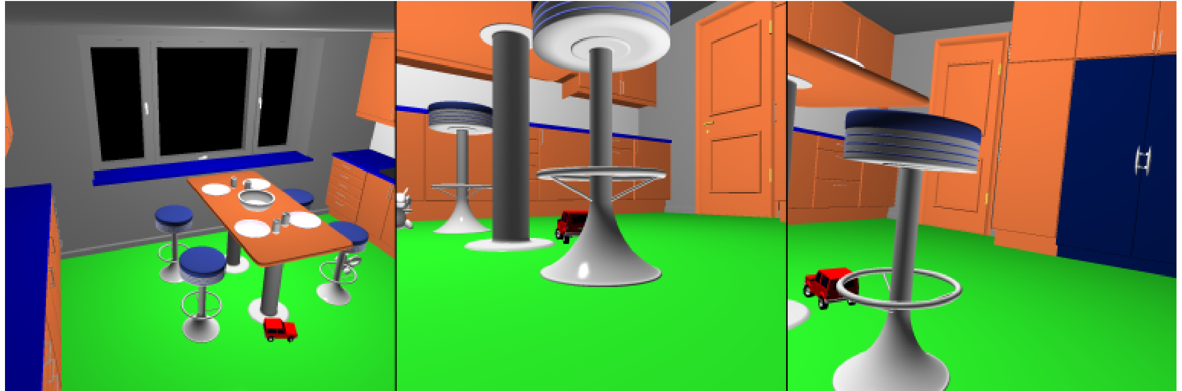
Figure 3.1: Stanford Models Render [33]

Model	Vertices	Triangles
Bunny	35947	69451
Dragon	566,098	1,132,830
Buddha	543,652	1,087,716

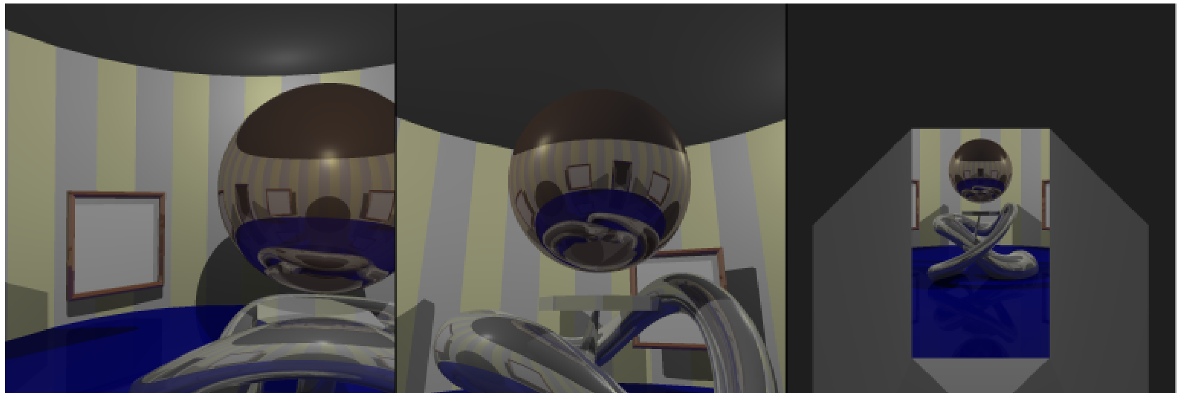
Table 3.1: Stanford Models - Vertex & Triangle Count [33]

3.1.2 BART

The BART (Benchmark for Animated Ray Tracing) scenes were created specifically to benchmark ray traced scenes. An example render of the BART scenes can be seen in figure 3.2 where the kitchen, museum and robot scenes can be seen. The paper "BART: A Benchmark for Animated Ray Tracing" describes the process of creating these models, the goal of the BART research was to identify what stresses ray tracing algorithms leading to a performance loss. They found a number of items that stressed a number of ray tracing implementations and took these into account when creating the BART models. [21]



(a) BART Kitchen Scene



(b) BART Museum Scene

(c) BART Robots Scene

Figure 3.2: BART Scenes [21]

3.1.3 MGF

The MGF models are another set of scenes that are often used for physically based rendering applications. The scenes I am using included the conference, office and theatre scenes which can be seen in figure 3.3.[17].

3.2 Traversal Time

The first of the experiments I will carry out will be to look at the traversal times of each acceleration data structure. In this experiment I am only interested in the time taken to traverse the acceleration data structure so I will used the CUDA 'clock' function to get these results, the clock function returns the current value of a counter that is incremented every clock cycle. [28] The 'clock' function will be invoked directly before traversing the structure and then directly after. I will then have two values the start clocks and the end clocks, using these values I can calculate the number of clocks that have passed by subtracting the start clocks from the end clocks. The result of this will give me the number of clock cycles passed for traversing an acceleration structure, but to get the time in milliseconds the number of clocks is divided by the speed of the GPU shader clock in kilohertz. An example of the code that will be used is shown in figure 3.4.

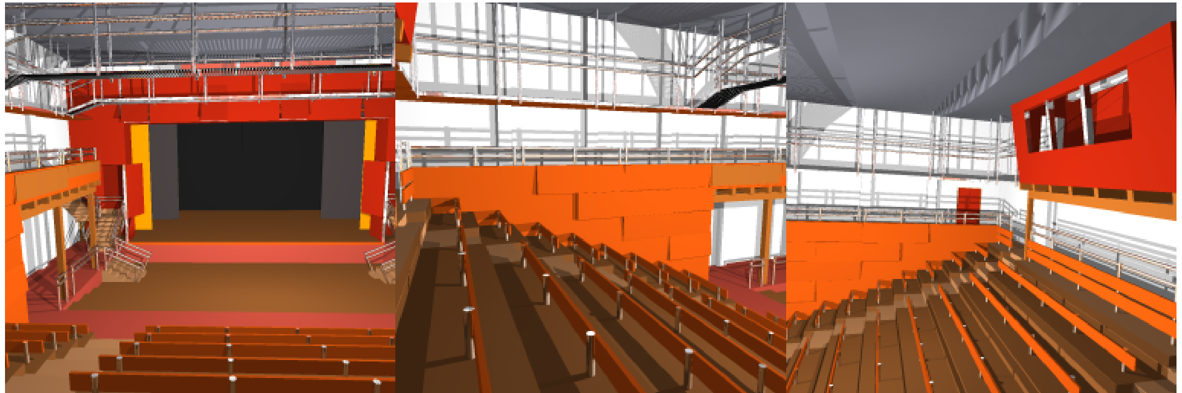
In the case of primary and shadow rays multiple traversals are required, in this case another variable is used to store the clocks passed after the first traversal, the results of the second traversal are added to this.

The traversal time experiment will be run using the Stanford, BART and MGF models for K-d trees and BVHs with primary and secondary rays. In the case of the BART and MGF scenes secondary rays will also be included in these results. The results of this experiment will give the min, max and average values for traversing these acceleration structures on a Fermi based GPU.

This experiment will also be run using a number of different cache configurations in order to see how the cache affects the traversal time, to carry out this experiment the

(a) MGF Conference Scene

(b) MGF Office Scene



(c) MGF Theatre Scene

Figure 3.3: MGF Scenes [17]

```
clock_t start = clock();

// Traverse the grid and try to find intersection.
int hitTriangle = traverseKdTree<true>(origin,dir,minimum,maximum,triangles,buffer,↩
    stackNode,stackMin,stackMax);

clock_t stop = clock();
```

Figure 3.4: Traversal Time Experiment

traversal will be run three times, the first with the default 16KB L1 with 48KB shared memory configuration, the second with 48KB L1 with 16KB shared memory and finally with the L1 cache completely disabled. The results of these will then be evaluated to see the traversal times.

3.3 Cache Performance

Making use of the NVidia Nsight developer tools I will be looking at the cache performance of these ray-tracing acceleration data structures. Fortunately NVidia provide a number of methods to change the size of the L1 cache as seen in figure 3.5, although the 32kb/32kb split is only available on cards based on the newer Kepler architecture. There is also an option to completely disable the L1 cache using a command line flag.

Value	**Description**
cudaFuncCachePreferNone	Default cache configuration (16kb L1 with 48kb shared) .
cudaFuncCachePreferShared	16kb L1 with 48kb shared.
cudaFuncCachePreferL1	48kb L1 with 16kb shared.
cudaFuncCachePreferEqual	32kb L1 with 32kb shared.

Figure 3.5: CUDA cache configuration options

This experiment will take place over one hundred frames for both primary rays and primary combined with shadow rays, the average hit rates across the one hundred frames will be the result.

3.3.1 16kb L1 with 48kb Shared Memory

This is the default setting used by the NVidia CUDA compiler (NVCC) which allocates 16kb to the level one cache and 48kb to shared memory.

3.3.2 48kb L1 with 16kb Shared Memory

There are two possible ways in which we can change the cache configuration. The first method involves adding the following command line arguments to NVCC - "-Xptxas -dlcm=ca", the second method involves adding a line of code as seen in figure 3.6.

```
cudaDeviceSetCacheConfig ( cudaFuncCachePreferL1 );
```

Figure 3.6: CUDA Cache Configuration

3.3.3 L1 Cache Disabled

The last cache experiment I plan to carry out involves completely disabling the L1 cache. When the L1 cache is disabled all global memory transactions that take place are 32 bytes in size (128 bytes when the L1 cache is enabled). To disable the L1 cache a command line flag is passed to the CUDA compiler as shown in figure 3.7.

```
-Xptxas -dlcm=cg
```

Figure 3.7: CUDA Disable L1 Cache

3.4 Branch Divergence

The branch divergence experiment looks at how many branches diverge over a number of kernel launches. This is important as conditional statements can lead to a performance decrease in a SM as each branch of the condition has to be evaluated. With a long code path conditionals can cause a two-times slowdown for each conditional inside

the warp, up to a maximum of a 32 times slowdown. With the Fermi architecture a new technique is utilised called prediction, with this technique the if and else parts of the conditional are executed in parallel which helps to solve the problem of mispredicted branches and warp divergence. [6]

The impact of branch divergence is that the SM units are underutilised which cannot be compensated by increasing the level of parallelism. With CUDA a kernel launch is executed on the GPU by scheduling thread blocks onto the streaming multiprocessors which must execute the thread block in its entirety. Each of these thread blocks is separated into groups of 32 threads which are known as warps, all threads inside this warp must execute the same instruction at any given time. In the case of branch divergence which causes different paths to be taken the warp will serially execute each branch path that is taken and disables threads that are not on that path, these threads then reconverge after the execution is complete. An example of potential branch divergence can be seen in figure 3.9 where if only half of the threads in a warp execute the '++x' statement then the utilisation of the execution units is only 50 percent. The paper "Reducing Branch Divergence in GPU Programs" presents two ways in which to avoid branch divergence which are referred to as iteration delaying and branch distribution. [11]

To get the branch statistics I plan to use the nvprof command line tool using the arguments shown in figure 3.8 for each model and all resolutions. To actually calculate the percentage of divergent branches I used the formula shown in figure 3.10.

```
nvprof --events branch,divergent_branch raytrace.exe
```

Figure 3.8: Branch Divergence nvprof

3.5 Instruction Statistics

The instruction statistics experiment looks at the number of instructions executed over a number of kernel launches. The results of this will be the minimum, maximum and

```
tid = threadIdx.x;
if (a[tid] > 0 ) {
  ++x;
}
```

Figure 3.9: Branch Divergence [11]

```
(100 * divergent branch) / (divergent branch + branch)
```

Figure 3.10: Divergent Branch Percentage [6]

average number of instructions executed for both K-d trees and BVHs, this experiment will be carried out using the Stanford, BART and MGF models.

Using the results of this experiment I will look to see the correlation between the number of instructions executed and the time it takes to traverse the acceleration data structures. Figure 3.11 shows how I will use nvprof command line tool to gather the data for this experiment.

```
nvprof --events inst_executed raytrace.exe
```

Figure 3.11: Instructions Executed with nvprof

3.6 Dynamic Scene

The dynamic scene experiment looks at the performance of K-d tree and BVH traversals when the data structures are rebuilt every frame, this experiment will be carried out over one hundred frames. For this experiment two different models are used, the Stanford bunny and the Stanford dragon in which these models move in the opposite direction to one another. The image in figure 3.12 shows an example of the motion in the dynamic scene where the two models move in different directions.

The results of this experiment will be looking at both the traversal and cache performance of the acceleration data structures for this dynamic scene. As with the previous experiments the results will be gathered using the CUDA 'clock' function to get the traversal times for each acceleration data structure. This experiment also looks at the cache performance of the dynamic scene looking at the L1 and L2 caches on the GPU, to gather these results NVidia's NSight tool will be used in a similar manner to that of the previous cache experiment.

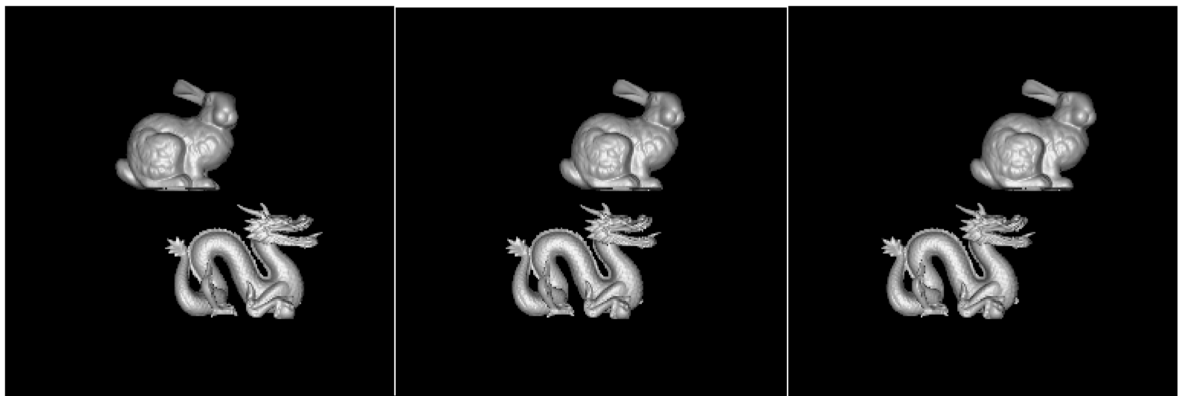


Figure 3.12: Dynamic Scene Motion

Chapter 4

Test Setup

The following chapter will discuss in detail the hardware setup I am using to carry out the experiments, the architecture of the GPU I am using (GTX 590) and the ray-tracing implementation used.

4.1 Hardware

The hardware setup used for this dissertation can be seen in table 4.1. I am using a Fermi based GPU which introduced a number of new features when it was released some of which will be discussed later in this chapter.

Component	**Value**
CPU	Intel i7-2600K @ 4.0GHz
GPU	NVidia GeForce GTX 590 (3GB)
Architecture	Fermi
Compute Capability	2.0
SLI	Disabled
RAM	16GB 1600MHz Dual Channel DDR3
OS	Windows 8 64-bit

Table 4.1: Hardware Setup

4.2 Tools

In order to get the required results for this dissertation I will be using a number of tools for profiling code that runs on the GPU. The tools I will be using to carry out the experiments are both from NVidia, nvprof and nsight. The use of profiling tools allow us to find bottlenecks in some code, in the case of GPUs they are best at parallel execution so if the occupancy rate is low that could be considered a bottleneck as the GPU is not fully utilised.

4.2.1 Nvprof

The nvprof tool is a command line program provided by NVidia with the CUDA GPU computing SDK which allows the user to measure a number of performance metrics. With the nvprof profiler we can look at data such as the number of instructions executed and the number of divergent branches. An alternative to the nvprof tool is the NVidia visual profiler which is essentially the visual version of this command line tool, with the use of this tool you can visually see the CPU timeline, GPU timeline, kernel properties along with others. [9] The full list of performance metrics available for the GTX 590 can be found in the appendix table A.1.

4.2.2 NSight

Another one of the tools I used for this dissertation is NVidia's Nsight debugging and profiling tool which is supported by both eclipse and visual studio. Nsight allows you to perform a number of experiments on a code base using hardware counters built into the GPU, this allows us to look at low-level information such as the cache hit rates. [2] A screenshot of the NVidia Nsight profiler can be seen in figure 4.1.

4.3 Fermi Architecture

In 2010 NVidia released their new series of graphics cards, the GeForce 400 series which were the first cards based on the new Fermi architecture. The Fermi architecture was a huge leap over the existing GT200 architecture with 3.0 billion transistors, more than twice the transistor count (1.4 billion) from the previous generation. This new

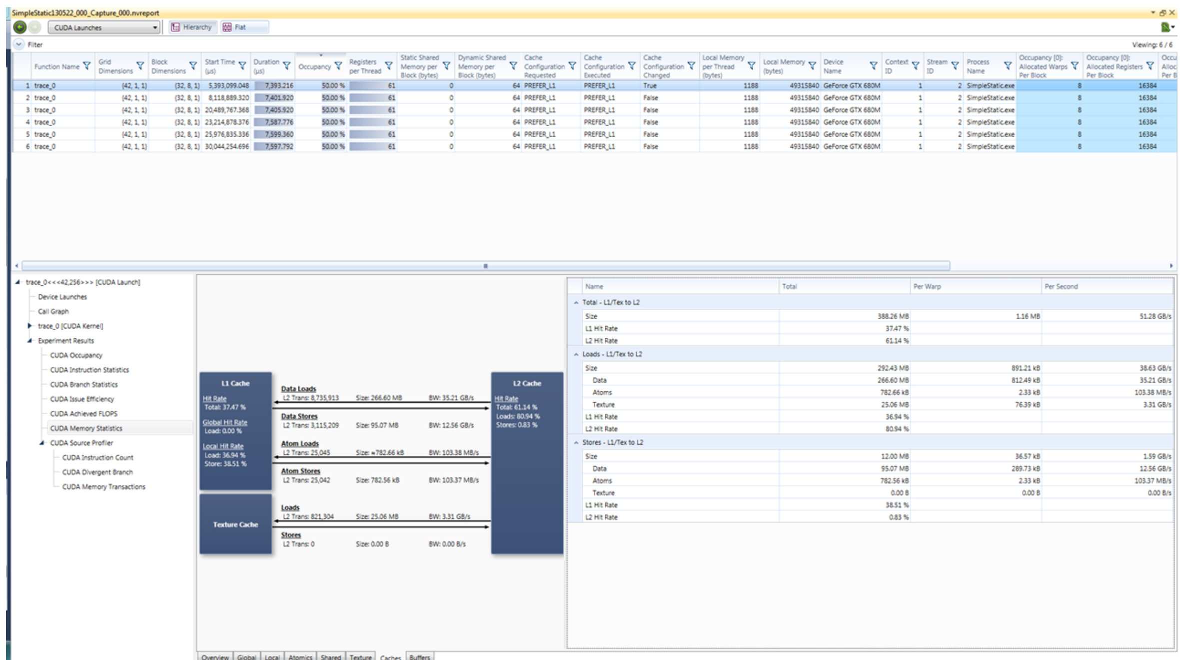


Figure 4.1: NVidia Nsight Profiler

architecture had a number of new features such as an increased core count for each streaming multiprocessor (SM), EEC memory support and configurable cache sizes along with many other features. [24]

4.3.1 Streaming Multiprocessors

GPUs based on the Fermi architecture are made up of 16 streaming multiprocessors (SM) each of which contains 32 CUDA cores giving 512 CUDA cores in total. All of the streaming multiprocessors share a common L2 cache which is 768kb in size, this can be seen in figure 4.2. An example of a streaming multiprocessor is shown in figure 4.3.

4.3.2 Configurable L1 / Shared Memory Cache

One of the new additions with the Fermi architecture is the ability to modify the size

of the L1/shared memory cache or even completely disable this cache. Each streaming multiprocessor (SM) has 64KB of on-chip memory. The default cache configuration allocates 16kb to the L1 cache and 48kb to shared memory, this can be changed to use a 48kb L1 and 16kb shared memory split or a 32kb/32kb even split between L1 and shared memory. This feature allows developers to gain performance improvements depending on how much shared memory they have used, for example if a developer only uses 8kb of shared memory they will likely gain a noticeable performance improvement by using the 16kb shared memory with 48kb L1 cache configuration. [4]

With the Fermi architecture memory transactions are either 32 bytes or 128 bytes in size. When the L1 cache is enabled then all memory transactions that take place are 128 bytes, in the case that the L1 cache is disabled memory transactions are 32 bytes. [36]

	Value
Transistors	3.0 billion
Streaming Multiprocessors (SMs)	16
Streaming Processors (Per-SM)	32
CUDA Cores	512
Registers (Per-SM)	32 KB
Max. number of threads (Per-SM)	1536
Max. number of threads (per-block)	1024
Warp size	32
Warp scheduler	Dual
Double Precision Floating Point Capability	256 FMA ops / clock
Single Precision Floating Point Capability	512 FMA ops / clock
Special Function Units (SFUs) / SM	4
Warp schedulers (per SM)	2
Shared Memory (per SM))	User Configurable (16, 32 or 48 KB)
L1 Cache (per SM))	User Configurable (0, 16, 32 or 48 KB)
L2 Cache	768k KB
Size of global memory transaction	32 or 128 B
ECC Memory Support	Yes
Concurrent Kernals	Up to 16
Load/Store Address Width	64-bit

Table 4.2: Fermi Specifications

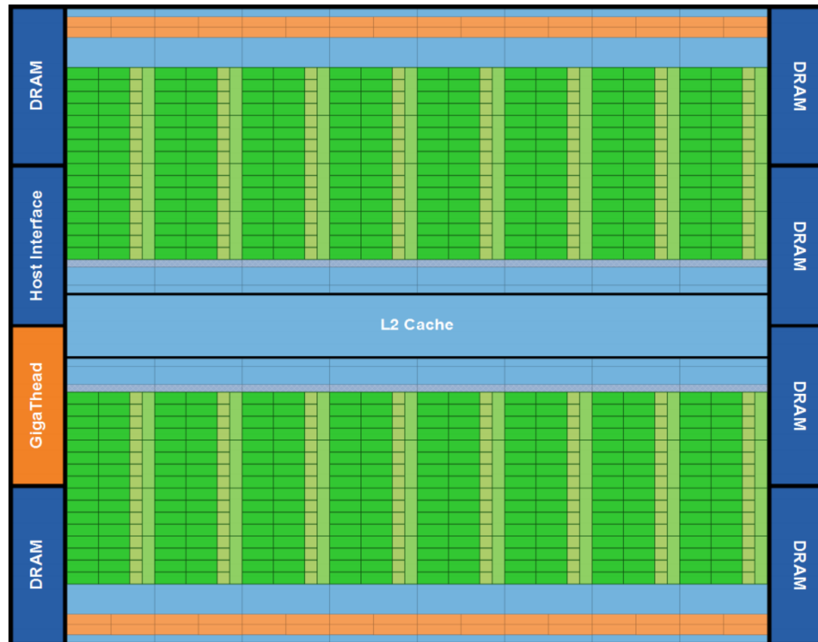


Figure 4.2: NVidia Fermi Architecture [24]

4.4 Implementation

I originally planned to use NVidia's Optix engine which supports a number of acceleration data structures such as K-d trees and BVHs, unfortunately the traversal code is closed source because of this I had to find another ray tracing implementation. After looking at various ray tracing code bases I decided to use the ray-tracing implementation from the paper "Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms". This code base has support for a number of acceleration structures such as K-d trees, BVHs and uniform grids though I will only be looking at K-d trees and BVHs, The actual code used in this implementation is C++ with CUDA. [38]

4.4.1 K-d Tree

The K-d tree used in this implementation is built using the surface area heuristic, each K-d node occupies 8 bytes of memory. The typical traversal for a K-d tree looks as in the figure 4.4 where the algorithm works by walking the tree until it finds a leaf node at which point it traverses the triangles at that node to find the closest hit. Unfortunately this does not scale well to the GPU so a number of techniques are employed such as

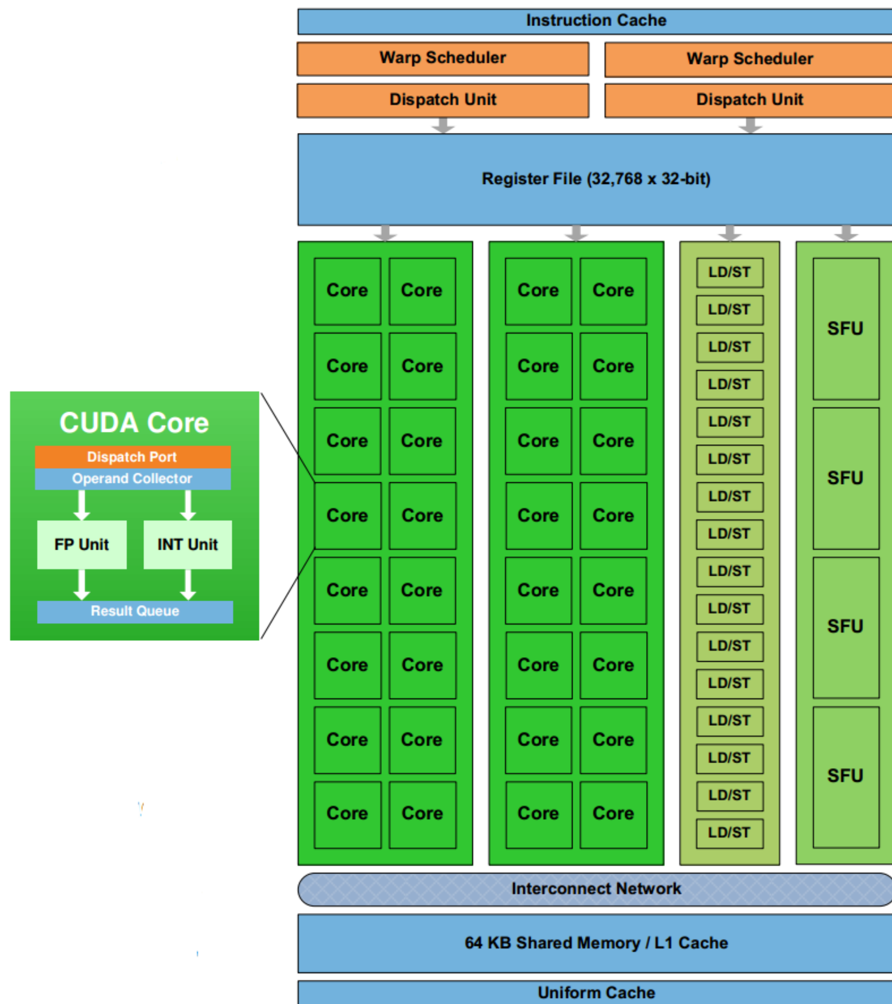


Figure 4.3: Streaming Multiprocessor Overview [24]

kd-restart, push-down and short-stack. [12]

```
stack.push(root,sceneMin,sceneMax)
tHit=infinity
while (not stack.empty()):
    (node,tMin,tMax)=stack.pop()
    while (not node.isLeaf()):
        a = node.axis
        tSplit = ( node.value − ray.origin[a] ) / ray.direction[a]
        (first, sec) = order(ray.direction[a], node.left, node.right)
        if( tSplit >= tMax or tSplit < 0)
            node=first
        else if( tSplit <= tMin)
            node=second
        else
            stack.push( sec, tSplit, tMax)
            node=first
            tMax=tSplit
    for tri in node.triangles():
        tHit=min(tHit,tri.Intersect(ray))
        if tHit<tMax:
            return tHit //early exit
return tHit
```

Figure 4.4: Typical K-d Traversal Psuedocode [12]

kd-restart

The kd-restart traversal algorithm changes the typical kd-tree traversal algorithm such that it eliminates any stack operations, by eliminating stack operations then the resulting traversal will move directly to the first leaf node that has been intersected by the ray. This means that if a ray leaves a leaf node without intersecting any of the triangles then the kd-restart algorithm advances the tMin and tMax values forward so that the origin of the ray (tMin) will be set to the previous endpoint, tMax. An example of this can be seen in figure 4.5. [7]

push-down

Another technique that is used is the push-down traversal algorithm which is used to prevent restarting traversal from the root node. Instead of restarting at the root node rays only need to backtrack to the node that is the root of the lowest subtree that encloses them. From an implementation point of view it is very easy to implement requiring just a boolean flag and some simple logic. [12]

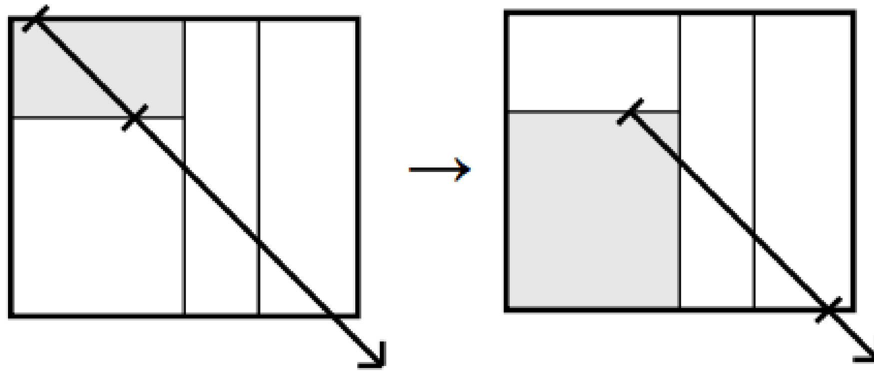


Figure 4.5: kd-restart Algorithm

short-stack

The short-stack is a technique that is complementary to the push-down technique. The short-stack introduces a small fixed-size stack with two modifications to how the stack is manipulated. The first modification is that when pushing a node onto the stack when the stack is full the bottom-most entry is discarded. The second modification is when an empty stack is popped a restart is triggered instead of terminating the traversal. [12]

4.4.2 BVH

The BVH tree in this implementation was built in a top-down manner using the surface area heuristic, the BVH cell occupies 32 bytes which is exactly four times more then a K-d tree node. The traversal algorithm used is based on the paper "Realtime Ray Tracing on GPU with BVH-based Packet Traversal". [8]

This BVH traversal algorithm uses a packet based approach which are essentially collections of coherent rays. Each ray is mapped to a single thread and each packet to a chunk which are aligned to 128 bytes on Fermi based GPUs.

The algorithm works by traversing one node at a time and testing an entire packet against it. If this node happens to be a leaf node then each ray in the packet is checked for an intersection with the geometry contained within the BVH node. If the current

node is not a leaf node then both child nodes are loaded and packet is tested against them to determine the traversal order, it does this by comparing the signed entry distance between the ray and the two child nodes to see which node the ray should traverse first. It is then up to the algorithm to decide which of the child nodes to visit, the algorithm looks at how many of the rays in the packet are trying to traverse each node and picks the node with the highest requested traversals, if a ray inside the packet wants to visit the other node then this node is pushed onto the stack. In the case that none of the rays contained in the packet want to visit these two child nodes or when the algorithm has just processed a leaf node the next node is taken from the top of the stack. If the stack is empty then the algorithm will terminate. Figure 4.6 shows the pseudocode for the BVH traversal algorithm. [10]

```
R = (O,D) // The ray
d ← ∞  // Distance to closest intersection
NP ← pointer to the BVH root

NL,NR : shared ≡ Shared storage for N's children
M[] : shared ≡ Reduction memory
S : shared ≡ The traversal stack
PID : const ≡ The number of this processor

loop
    if NP points to a leaf then
        Intersect R with contained geometry
        Update d if necessary
        break, if S is empty
        NP ← pop(S)
    else
        if PID < size(NL,NR) then // parallel read
            (NL,NR)[PID] ← children(NP)[PID]
        end if

        (λ1,λ2) ← intersect(R,NL)
        (μ1,μ2) ← intersect(R,NR)
        b1 ← (λ1 < λ2) ∧ (λ1 < d) ∧ (λ2 ≥ 0)
        b2 ← (μ1 < μ2) ∧ (μ1 < d) ∧ (μ2 ≥ 0)

        M[PID] ← false, if PID < 4
        M[2b1 +b2] ← true
        if M[3] ∨ M[1] ∧ M[2] then // Visit both children
            M[PID] ← 2(b2 ∧ μ1 < λ1)   1
            PARALLELSUM(M[0 .. processor−count])

            if M[0] < 0 then
                (NN,NF) ← pointer−to (NL,NR)
            else
                (NN,NF) ← pointer−to (NR,NL)
            end if

            push(S,NF), if PID = 0
            NP ← NN
        else if M[1] then
            NP ← pointer−to(NL)
        else if M[2] then
            NP ← pointer−to(NR)
        else
            break, if S is empty
            NP ← pop(S)
        end if
    end if
end loop
```

Figure 4.6: GPU BVH Traversal Psuedocode [12]

Chapter 5

The Results

In this chapter the results of the experiments described in the 'Method' chapter are presented. The results were compiled using a number of tools both from NVidia, nvprof and NSight.

5.1 Full Render

In this section the results of the full render experiment are presented. This experiment looks at the entire kernel launch including both the traversal of the acceleration data structure and shading resulting in the time it takes to render a single frame.

5.1.1 Stanford

The results of the full render experiment when carried out using the Stanford models show that in most cases the K-d tree performs better than the BVH (With the exception of the Stanford bunny) at the lower resolutions (256x256, 512x512 and 768x768) but the BVH closes this performance gap at the higher resolution of 1024x1024. The results of this experiment can been seen below in tables 5.1 and 5.2.

	Min	**Max**	**Average**
K-d Bunny P			
256x256	1.072832 ms	2.407136 ms	1.548652 ms
512x512	2.431680 ms	4.832672 ms	3.343905 ms
768x768	4.188160 ms	7.550336 ms	5.780483 ms
1024x1024	6.263360 ms	11.41248 ms	8.828431 ms
BVH Bunny P			
256x256	1.084544 ms	2.069984 ms	1.454210 ms
512x512	1.560864 ms	2.424480 ms	1.953591 ms
768x768	2.460704 ms	3.656160 ms	2.945168 ms
1024x1024	3.469792 ms	5.261280 ms	4.275245 ms
K-d Dragon P			
256x256	1.305312 ms	3.062720 ms	1.991542 ms
512x512	3.292064 ms	6.270080 ms	4.709790 ms
768x768	5.923712 ms	11.447104 ms	8.491594 ms
1024x1024	9.152352 ms	17.964449 ms	13.191978 ms
BVH Dragon P			
256x256	2.670976 ms	8.619104 ms	4.671336 ms
512x512	3.966944 ms	9.686464 ms	7.008855 ms
768x768	4.986848 ms	12.602784 ms	9.154868 ms
1024x1024	6.495200 ms	15.592224 ms	11.537935 ms
K-d Buddha P			
256x256	0.907648 ms	2.501632 ms	1.592877 ms
512x512	1.940128 ms	5.024800 ms	3.671167 ms
768x768	3.280704 ms	8.659232 ms	6.527819 ms
1024x1024	5.062336 ms	13.258944 ms	10.047005 ms
BVH Buddha P			
256x256	1.948224 ms	7.836416 ms	4.310605 ms
512x512	2.603424 ms	9.286048 ms	6.403870 ms
768x768	3.529280 ms	11.278240 ms	8.397223 ms
1024x1024	4.423040 ms	13.877952 ms	10.433164 ms

P Primary Rays

Table 5.1: Stanford Primary Ray Full Render

	Min	Max	Average
K-d Bunny PS			
256x256	2.725504 ms	3.514688 ms	2.479525 ms
512x512	4.676032 ms	8.146400 ms	5.859196 ms
768x768	8.559776 ms	14.501120 ms	10.796020 ms
1024x1024	12.874944 ms	22.158049 ms	16.663357 ms
BVH Bunny PS			
256x256	1.981984 ms	4.258560 ms	2.742734 ms
512x512	3.235616 ms	4.278752 ms	3.640488 ms
768x768	5.037856 ms	6.632704 ms	5.753088 ms
1024x1024	7.396320 ms	9.547520 ms	8.444658 ms
K-d Dragon PS			
256x256	1.305312 ms	4.751520 ms	3.391940 ms
512x512	7.007840 ms	11.342944 ms	8.868724 ms
768x768	13.464544 ms	21.814816 ms	16.745096 ms
1024x1024	20.817345 ms	33.831841 ms	26.078743 ms
BVH Dragon PS			
256x256	6.662656 ms	10.410912 ms	8.489571 ms
512x512	11.555072 ms	17.183071 ms	14.102702 ms
768x768	16.158144 ms	22.065825 ms	19.581047 ms
1024x1024	20.798912 ms	30.153215 ms	25.040182 ms
K-d Buddha PS			
256x256	2.048320 ms	3.921440 ms	2.883934 ms
512x512	4.687520 ms	8.698368 ms	7.099356 ms
768x768	8.460064 ms	15.561088 ms	12.903708 ms
1024x1024	13.200448 ms	24.662560 ms	20.106138 ms
BVH Buddha PS			
256x256	6.083936 ms	11.026464 ms	8.433007 ms
512x512	10.857024 ms	17.850401 ms	14.648832 ms
768x768	14.578624 ms	22.863647 ms	19.644007 ms
1024x1024	18.544704 ms	27.985472 ms	24.548191 ms

PS Primary & Shadow Rays

Table 5.2: Stanford Primary & Shadow Rays Full Render

5.1.2 BART

When using the BART set of models consisting of the robots, museum and kitchen scenes there is no clear winner with regards to the performance of the acceleration data structures. The results of this experiment can been seen in tables 5.3, 5.4 and 5.5 below.

In the first case we look at the performance of the robots scene and in this particular case the K-d tree is faster in all cases (primary, primary with shadow and primary, shadow and reflection rays).

In the second case we look at the museum scene, in the museum scene the BVH is faster in all of the tests and close to 50% faster in certain cases.

Lastly we look at the performance of the kitchen scene, this scene has quite a mixed performance where the BVH is faster than the K-d tree with just primary rays while the K-d tree outperforms the BVH with reflection rays. In the kitchen scene with primary and shadow rays the K-d tree performs better at 256x256 and 512x512 while the BVH performs better at 768x768 and 1024x1024.

Robots	Min	Max	Average
K-d P			
256x256	0.303712 ms	2.163328 ms	1.111406 ms
512x512	1.118336 ms	4.959776 ms	2.646239 ms
768x768	2.752544 ms	13.583232 ms	7.168683 ms
1024x1024	4.635808 ms	18.738752 ms	10.458248 ms
BVH P			
256x256	0.474752 ms	2.636512 ms	1.367954 ms
512x512	1.767712 ms	4.485888 ms	3.294106 ms
768x768	4.374816 ms	11.827392 ms	8.348372 ms
1024x1024	7.364160 ms	18.528511 ms	13.342626 ms
K-d PS			
256x256	0.696832 ms	3.348640 ms	1.922398 ms
512x512	2.625856 ms	8.737024 ms	5.050729 ms
768x768	6.396864 ms	22.179968 ms	13.132985 ms
1024x1024	10.838272 ms	31.304064 ms	19.817677 ms
BVH PS			
256x256	0.834688 ms	5.065024 ms	2.557266 ms
512x512	3.099104 ms	8.409472 ms	6.140616 ms
768x768	7.904192 ms	22.808640 ms	15.562988 ms
1024x1024	13.019840 ms	34.423328 ms	24.853336 ms
K-d PSR			
256x256	0.703840 ms	4.721760 ms	2.613482 ms
512x512	2.640352 ms	10.989344 ms	5.788201 ms
768x768	6.431360 ms	34.955841 ms	16.052902 ms
1024x1024	10.917248 ms	45.430782 ms	22.98251 ms
BVH PSR			
256x256	0.883296 ms	8.036512 ms	3.935138 ms
512x512	3.313408 ms	12.906176 ms	7.761605 ms
768x768	8.559424 ms	43.155968 ms	21.005152 ms
1024x1024	14.225280 ms	58.282497 ms	30.675678 ms

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.3: BART Robots Traversal Time K-d tree and BVH

Muesum	**Min**	**Max**	**Average**
K-d P			
256x256	0.595168 ms	2.966496 ms	2.121926 ms
512x512	1.771296 ms	6.576896 ms	5.01249 ms
768x768	4.284736 ms	17.886656 ms	13.539915 ms
1024x1024	6.965088 ms	24.139040 ms	18.588215 ms
BVH P			
256x256	0.440320 ms	1.234016 ms	1.008350 ms
512x512	1.593888 ms	3.126848 ms	2.610556 ms
768x768	3.917728 ms	7.617664 ms	6.345273 ms
1024x1024	6.565248 ms	11.931008 ms	9.988448 ms
K-d PS			
256x256	2.180192 ms	5.455680 ms	4.497135 ms
512x512	6.607328 ms	14.585632 ms	12.199412 ms
768x768	15.407072 ms	36.597279 ms	29.912104 ms
1024x1024	25.110624 ms	51.801922 ms	43.964321 ms
BVH PS			
256x256	1.382016 ms	3.255040 ms	2.806842 ms
512x512	4.409024 ms	8.080896 ms	6.969739 ms
768x768	10.743648 ms	18.775520 ms	16.552038 ms
1024x1024	17.816193 ms	29.201281 ms	25.657625 ms
K-d PSR			
256x256	2.739776 ms	11.646976 ms	9.258921 ms
512x512	7.767072 ms	30.527617 ms	24.021410 ms
768x768	18.512224 ms	77.097092 ms	60.453091 ms
1024x1024	29.225697 ms	104.383774 ms	86.116585 ms
BVH PSR			
256x256	1.816928 ms	12.334912 ms	10.142564 ms
512x512	5.348832 ms	24.822496 ms	20.053637 ms
768x768	13.796896 ms	65.077087 ms	51.375050 ms
1024x1024	22.012928 ms	85.104736 ms	67.532257 ms

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.4: BART Museum Traversal Time K-d tree and BVH

Kitchen	Min	Max	Average
K-d P			
256x256	0.168320 ms	3.463552 ms	2.343429 ms
512x512	0.619104 ms	6.050944 ms	4.333699 ms
768x768	1.558848 ms	20.482529 ms	12.587105 ms
1024x1024	2.594432 ms	24.238625 ms	16.367794 ms
BVH P			
256x256	0.561632 ms	2.485696 ms	1.626423 ms
512x512	2.129120 ms	4.937440 ms	3.848129 ms
768x768	5.231264 ms	12.040800 ms	9.638050 ms
1024x1024	8.874976 ms	17.859585 ms	14.620613 ms
K-d PS			
256x256	1.699200 ms	9.2840000 ms	6.471941 ms
512x512	6.087904 ms	25.288063 ms	17.370169 ms
768x768	14.503360 ms	66.022881 ms	44.233585 ms
1024x1024	24.597919 ms	93.149117 ms	64.363754 ms
BVH PS			
256x256	2.823552 ms	9.540672 ms	6.686491 ms
512x512	10.742688 ms	23.038624 ms	17.694319 ms
768x768	27.585567 ms	52.916767 ms	41.969330 ms
1024x1024	45.775520 ms	79.503136 ms	64.508408 ms
K-d PSR			
256x256	1.723072 ms	15.293760 ms	10.312797 ms
512x512	6.205952 ms	41.103489 ms	25.812811 ms
768x768	14.777888 ms	119.055809 ms	70.401917 ms
1024x1024	25.146303 ms	163.017380 ms	98.886032 ms
BVH PSR			
256x256	2.898176 ms	30.198816 ms	17.988287 ms
512x512	11.063456 ms	71.271713 ms	37.457893 ms
768x768	29.745920 ms	201.298141 ms	108.049805 ms
1024x1024	49.096802 ms	262.157043 ms	143.415359 ms

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.5: BART Kitchen Traversal Time K-d tree and BVH

5.1.3 MGF

In the case of the MGF scenes the BVH has the upper hand over the K-d tree. The results of this experiment can be seen in tables 5.6, 5.7 and 5.8.

The first scene tested was the theatre scene. In the case of the theatre scene the BVH was slighty faster than the K-d tree with primary rays and primary with shadow rays by around one to two milliseconds. Although the K-d tree outperforms the BVH in the case of reflection rays where it was three to thirty milliseconds faster in cases.

The second scene tested was the conference scene. The results of this experiment carried out on the conference scene shows that the BVH is faster than the K-d tree in all cases and in some particular cases it is around two to three times faster than the K-d tree.

Finally, the last scene tested was the office scene. The results for the office scene show that the BVH was faster than the K-d tree in all cases ranging from a three to fifteen millisecond difference between the acceleration data structures.

Theatre	Min	Max	Average
K-d P			
256x256	0.359488 ms	4.185504 ms	2.103928 ms
512x512	1.275776 ms	10.313184 ms	5.286815 ms
768x768	3.150528 ms	25.217888 ms	13.267407 ms
1024x1024	5.178400 ms	36.528255 ms	19.298046 ms
BVH P			
256x256	0.691648 ms	2.631456 ms	1.421610 ms
512x512	2.446048 ms	9.240000 ms	4.521169 ms
768x768	6.020064 ms	22.316704 ms	10.907532 ms
1024x1024	10.231744 ms	37.817089 ms	17.551369 ms
K-d PS			
256x256	2.443648 ms	8.281088 ms	4.774881 ms
512x512	6.556448 ms	19.836832 ms	13.002388 ms
768x768	16.285088 ms	47.485474 ms	31.193853 ms
1024x1024	26.262272 ms	70.616386 ms	47.151222 ms
BVH PS			
256x256	2.048288 ms	6.458112 ms	4.018600 ms
512x512	7.184192 ms	19.946592 ms	12.202060 ms
768x768	17.559904 ms	48.062366 ms	28.947918 ms
1024x1024	29.530144 ms	77.946274 ms	45.793098 ms
K-d PSR			
256x256	2.392192 ms	17.299168 ms	8.270072 ms
512x512	8.210048 ms	43.219456 ms	20.526312 ms
768x768	19.616896 ms	117.579262 ms	58.375954 ms
1024x1024	32.408417 ms	155.701981 ms	80.660133 ms
BVH PSR			
256x256	2.715552 ms	21.389185 ms	11.131578 ms
512x512	9.964608 ms	54.639137 ms	26.712067 ms
768x768	29.745920 ms	58.375954 ms	84.549545 ms
1024x1024	39.511230 ms	191.595490 ms	113.011917 ms

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.6: MGF Theatre Traversal Time K-d tree and BVH

Conference	Min	Max	Average
K-d P			
256x256	2.994656 ms	4.638560 ms	3.889329 ms
512x512	5.704544 ms	10.302048 ms	8.073959 ms
768x768	15.414368 ms	37.527519 ms	29.575521 ms
1024x1024	19.719584 ms	44.459263 ms	35.089020 ms
BVH P			
256x256	1.268640 ms	2.859328 ms	1.881490 ms
512x512	2.685376 ms	4.461664 ms	3.701239 ms
768x768	7.549728 ms	12.125120 ms	10.358947 ms
1024x1024	10.443648 ms	16.117376 ms	13.963135 ms
K-d PS			
256x256	6.183744 ms	8.889824 ms	7.628999 ms
512x512	11.346048 ms	20.624001 ms	16.823334 ms
768x768	31.588863 ms	67.459549 ms	54.038280 ms
1024x1024	41.038433 ms	81.769089 ms	67.061821 ms
BVH PS			
256x256	3.813280 ms	8.575680 ms	5.677174 ms
512x512	7.334560 ms	13.087168 ms	10.644398 ms
768x768	20.415104 ms	34.553055 ms	29.301922 ms
1024x1024	28.677889 ms	45.029728 ms	39.017368 ms
K-d PSR			
256x256	N/A	N/A	N/A
512x512	N/A	N/A	N/A
768x768	N/A	N/A	N/A
1024x1024	N/A	N/A	N/A
BVH PSR			
256x256	6.349600 ms	13.411200 ms	9.467188 ms
512x512	8.523904 ms	18.216703 ms	13.155451 ms
768x768	23.713535 ms	55.405281 ms	42.685680 ms
1024x1024	31.760736 ms	66.239487 ms	51.851124 ms

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.7: MGF Conference Traversal Time K-d tree and BVH

Office	Min	Max	Average
K-d P			
256x256	2.282336 ms	3.438752 ms	2.678899 ms
512x512	4.125344 ms	5.503456 ms	4.672025 ms
768x768	11.973952 ms	14.977952 ms	13.265472 ms
1024x1024	14.605024 ms	18.986784 ms	16.471052 ms
BVH P			
256x256	1.063680 ms	1.638720 ms	1.262682 ms
512x512	2.371904 ms	2.953056 ms	2.638733 ms
768x768	7.264320 ms	8.994880 ms	8.214394 ms
1024x1024	8.887424 ms	11.415008 ms	10.377632 ms
K-d PS			
256x256	5.675968 ms	8.383072 ms	6.899347 ms
512x512	13.032224 ms	17.562241 ms	15.666132 ms
768x768	37.140610 ms	42.789185 ms	39.395340 ms
1024x1024	51.370304 ms	57.711777 ms	54.566345 ms
BVH PS			
256x256	3.628256 ms	5.159360 ms	4.313031 ms
512x512	9.080192 ms	11.031584 ms	10.292833 ms
768x768	25.479391 ms	29.543488 ms	28.032146 ms
1024x1024	35.647488 ms	40.958206 ms	38.808231 ms
K-d PSR			
256x256	5.654048 ms	8.184576 ms	6.791181 ms
512x512	12.991168 ms	17.245216 ms	15.525816 ms
768x768	36.954815 ms	43.668831 ms	39.612053 ms
1024x1024	51.617825 ms	58.371391 ms	54.614952 ms
BVH PSR			
256x256	3.752320 ms	5.400032 ms	4.456864 ms
512x512	9.426848 ms	11.289280 ms	10.584877 ms
768x768	26.543009 ms	30.921600 ms	28.934631 ms
1024x1024	36.927551 ms	42.186081 ms	39.981724 ms

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.8: MGF Office Traversal Time K-d tree and BVH

5.2 Traversal Time Per Pixel

The traversal time per pixel experiment looks at the time taken to traverse the acceleration data structure for a single pixel (or ray). The results of this experiment show the average time to traverse the acceleration data structure for a number of different cache configurations.

5.2.1 Stanford

The results of this experiment carried out using the Stanford models shows that the K-d tree gains a performance increase by up to just under twelve percent by increasing the size of the L1 cache. In the case of the BVH increasing the size of the L1 cache has a negative result in most cases. When the L1 cache is completely disabled we see that there is a performance loss in the majority of cases.

In the case of the Stanford bunny the BVH outperforms the bunny for both primary and primary with shadow rays, even with the K-d's performance gain from the increased cache size it is around 75% slower than the BVH.

Looking at the Stanford dragon the results show that the K-d tree performs better than the BVH in most cases, but it is worth noting that the difference in traversal time becomes smaller and smaller as the resolution is increased. In one case with primary rays at 1024x1024 the BVH is faster than the K-d tree and with primary with shadow rays it becomes close to outpeforming the K-d tree.

Finally, the last of the Stanford models is the Stanford buddha. In this case again the K-d tree outperforms the BVH, but we see the same pattern again where the gap between the K-d tree and BVH narrow as the resolution increases.

Bunny	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	1.22875 ms	1.17430 ms (+4.431%)	1.27880 ms (-4.073%)
512x512	1.56195 ms	1.40837 ms (+9.833%)	1.54535 ms (-1.063%)
768x768	1.55545 ms	1.53659 ms (+1.212%)	1.69054 ms (-8.685%)
1024x1024	1.68890 ms	1.62293 ms (+3.906%)	1.78997 ms (-5.984%)
BVH P			
256x256	1.16257 ms	1.15950 ms (+0.264%)	1.20787 ms (-3.897%)
512x512	0.79086 ms	0.78968 ms (+0.149%)	0.80910 ms (-2.306%)
768x768	0.58691 ms	0.64536 ms (-9.959%)	0.65780 ms (-10.777%)
1024x1024	0.54383 ms	0.57512 ms (-5.754%)	0.58442 ms (-7.464%)
K-d PS			
256x256	1.72636 ms	1.56922 ms (+9.102%)	1.71973 ms (+0.384%)
512x512	1.88288 ms	1.64852 ms (+12.447%)	1.81031 ms (+3.854%)
768x768	1.87750 ms	1.72407 ms (+8.172%)	1.89443 ms (-0.902%)
1024x1024	2.03498 ms	1.79873 ms (+11.609%)	1.97789 ms (+2.805%)
BVH PS			
256x256	2.68731 ms	2.67354 ms (+0.512%)	2.76183 ms (-2.773%)
512x512	1.64260 ms	1.63110 ms (+0.700%)	1.66552 ms (-1.395%)
768x768	1.15711 ms	1.23090 ms (-6.377%)	1.24582 ms (-7.667%)
1024x1024	1.02148 ms	1.03442 ms (-1.267%)	1.04732 ms (-2.530%)

P Primary Rays

PS Primary & Shadow Rays

Table 5.9: Stanford Bunny Average Traversal Times (Per Pixel)

Dragon	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	1.47826 ms	1.42164 ms (+3.830%)	1.50828 ms (-2.031%)
512x512	1.91339 ms	1.73712 ms (+9.212%)	1.87432 ms (+2.042%)
768x768	1.95931 ms	1.92022 ms (+1.995%)	2.09122 ms (-6.732%)
1024x1024	2.16546 ms	2.04579 ms (+5.526%)	2.23100 ms (-3.027%)
BVH P			
256x256	3.47106 ms	3.46517 ms (+0.170%)	3.58479 ms (-3.277%)
512x512	2.77253 ms	2.76660 ms (+0.214%)	2.85024 ms (-2.803%)
768x768	2.02301 ms	2.23469 ms (-10.464%)	2.29374 ms (-13.383%)
1024x1024	1.74730 ms	1.87446 ms (-7.278%)	1.92529 ms (-10.187%)
K-d PS			
256x256	1.95353 ms	1.82236 ms (+6.715%)	1.96453 ms (-0.563%)
512x512	2.30247 ms	2.02740 ms (+11.947%)	2.21581 ms (+3.764%)
768x768	2.24419 ms	2.11847 ms (+5.602%)	2.31723 ms (-3.255%)
1024x1024	2.36666 ms	2.16550 ms (+8.500%)	2.36606 ms (+0.025%)
BVH PS			
256x256	5.34768 ms	5.32514 ms (+0.422%)	5.48009 ms (-2.476%)
512x512	3.64548 ms	3.62840 ms (+0.469%)	3.73831 ms (-2.546%)
768x768	2.78200 ms	2.98684 ms (-7.363%)	3.07704 ms (-10.605%)
1024x1024	2.49699 ms	2.59533 ms (-3.938%)	2.66413 ms (-6.693%)

P	Primary Rays
PS	Primary & Shadow Rays

Table 5.10: Stanford Dragon Average Traversal Times (Per Pixel)

Buddha	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	1.27154 ms	1.22827 ms (+3.403%)	1.30133 ms (-2.343%)
512x512	1.66757 ms	1.53381 ms (+8.021%)	1.64974 ms (+1.069%)
768x768	1.73071 ms	1.71681 ms (+0.803%)	1.86330 ms (-7.661%)
1024x1024	1.90991 ms	1.83504 ms (+3.920%)	2.00027 ms (-4.731%)
BVH P			
256x256	3.51910 ms	3.51704 ms (+0.059%)	3.64502 ms (-3.578%)
512x512	3.00823 ms	2.99752 ms (+0.356%)	3.09171 ms (-2.775%)
768x768	2.28501 ms	2.51972 ms (-10.272%)	2.59470 ms (-13.553%)
1024x1024	2.00215 ms	2.16746 ms (-8.257%)	2.22810 ms (-11.285%)
K-d PS			
256x256	1.86477 ms	1.75909 ms (+5.667%)	1.8813 ms (-0.886%)
512x512	2.19201 ms	1.98930 ms (+9.248%)	2.1558 ms (+1.652%)
768x768	2.16434 ms	2.12295 ms (+1.912%)	2.3154 ms (-6.979%)
1024x1024	2.33132 ms	2.20390 ms (+5.466%)	2.4112 ms (-3.426%)
BVH PS			
256x256	5.82499 ms	5.79936 ms (+0.440%)	5.99704 ms (-2.954%)
512x512	5.03099 ms	5.00617 ms (+0.493%)	5.16063 ms (-2.577%)
768x768	4.06612 ms	4.48649 ms (-10.338%)	4.62450 ms (-13.733%)
1024x1024	3.51192 ms	3.78270 ms (-7.710%)	3.89004 ms (-10.767%)

P Primary Rays

PS Primary & Shadow Rays

Table 5.11: Stanford Buddha Average Traversal Times (Per Pixel)

5.2.2 BART

Similarly to the experiment carried out with the Stanford models increasing the size of the L1 cache results in a performance increase of just over 16% in cases. While with the BVH we gain a performance increase of around 1% in cases while losing performance in other cases. With the L1 cache disabled there is a performance loss in all cases.

In the BART kitchen scene the K-d tree has a better average traversal time than the BVH. In the full render experiment we see that the BVH is faster with primary rays, this is not the case here which means that the K-d tree has a higher average maximum traversal time.

With the BART museum scene we can see that the BVH is faster in most cases than the K-d tree with exception of the primary rays experiment where they are very close to each other (within 8 thousandths of a millisecond in the worst case scenario).

The last scene in the BART set of models is the robots scene. In this scene we can see that the average traversal time of the K-d tree is faster in all cases which is similar to what we see in the full render experiment.

Kitchen	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	0.1223 ms	0.1083 ms (13.0057%)	0.1283 ms (-4.6538%)
512x512	0.0985 ms	0.0862 ms (14.3812%)	0.1027 ms (-4.012%)
768x768	0.0881 ms	0.0767 ms (14.8696%)	0.0915 ms (-3.6668%)
1024x1024	0.0819 ms	0.0712 ms (15.1462%)	0.0848 ms (-3.3629%)
BVH P			
256x256	0.1371 ms	0.1367 ms (0.3269%)	0.1411 ms (-2.8423%)
512x512	0.1066 ms	0.1063 ms (0.2492%)	0.1103 ms (-3.3480%)
768x768	0.0973 ms	0.0972 ms (0.2045%)	0.1010 ms (-3.5423%)
1024x1024	0.0930 ms	0.0929 ms (0.1769%)	0.0966 ms (-3.6326%)
K-d PS			
256x256	0.5813 ms	0.5102 ms (13.9651%)	0.6060 ms (-4.0548%)
512x512	0.4766 ms	0.4143 ms (15.0296%)	0.4937 ms (-3.4596%)
768x768	0.4299 ms	0.3723 ms (15.4934%)	0.4439 ms (-3.1373%)
1024x1024	0.4025 ms	0.3478 ms (15.7336%)	0.4144 ms (-2.8664%)
BVH PS			
256x256	0.7136 ms	0.7048 ms (1.2551%)	0.7358 ms (-3.0069%)
512x512	0.5324 ms	0.5269 ms (1.0543%)	0.5519 ms (-3.5141%)
768x768	0.4754 ms	0.4712 ms (0.8989%)	0.4942 ms (-3.8054%)
1024x1024	0.4478 ms	0.4443 ms (0.8029%)	0.4663 ms (-3.9601%)
K-d PSR			
256x256	0.7539 ms	0.6626 ms (13.7861%)	0.7884 ms (-4.3641%)
512x512	0.6329 ms	0.5510 ms (14.8892%)	0.6586 ms (-3.8963%)
768x768	0.5752 ms	0.4983 ms (15.4376%)	0.5966 ms (-3.5862%)
1024x1024	0.5400 ms	0.4663 ms (15.8160%)	0.5587 ms (-3.3392%)
BVH PSR			
256x256	1.4103 ms	1.3931 ms (1.2345%)	1.4491 ms (-2.6721%)
512x512	1.0283 ms	1.0170 ms (1.1167%)	1.0584 ms (-2.8474%)
768x768	0.8755 ms	0.8663 ms (1.0670%)	0.9028 ms (-3.0189%)
1024x1024	0.7909 ms	0.7829 ms (1.0209%)	0.8166 ms (-3.1434%)

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.12: BART Kitchen Average Traversal Times (Per Pixel)

Museum	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	0.0288 ms	0.0255 ms (13.1344%)	0.0298 ms (-3.4362%)
512x512	0.0264 ms	0.0243 ms (8.3519%)	0.0276 ms (-4.626%)
768x768	0.0256 ms	0.0237 ms (8.1135%)	0.0265 ms (-3.1399%)
1024x1024	0.0253 ms	0.0234 ms (8.2505%)	0.0262 ms (-3.3977%)
BVH P			
256x256	0.0280 ms	0.0279 ms (0.2976%)	0.0294 ms (-4.6644%)
512x512	0.0265 ms	0.0265 ms (0.1077%)	0.028 ms (-5.2511%)
768x768	0.0260 ms	0.0259 ms (0.0317%)	0.0275 ms (-5.7433%)
1024x1024	0.0255 ms	0.0256 ms (-0.2232%)	0.0271 ms (-5.8425%)
K-d PS			
256x256	0.0946 ms	0.0831 ms (13.8159%)	0.0969 ms (-2.4335%)
512x512	0.0895 ms	0.0786 ms (13.8247%)	0.0918 ms (-2.5604%)
768x768	0.0856 ms	0.0773 ms (10.6823%)	0.089 ms (-3.8758%)
1024x1024	0.0849 ms	0.0772 ms (9.9213%)	0.0876 ms (-3.174%)
BVH PS			
256x256	0.0693 ms	0.0697 ms (-0.5704%)	0.0737 ms (-5.9254%)
512x512	0.0668 ms	0.0668 ms (-0.0057%)	0.0711 ms (-5.9597%)
768x768	0.0643 ms	0.0644 ms (-0.1823%)	0.0689 ms (-6.6526%)
1024x1024	0.0633 ms	0.0632 ms (0.1585%)	0.0675 ms (-6.2519%)
K-d PSR			
256x256	0.1040 ms	0.0896 ms (16.1377%)	0.1063 ms (-2.16%)
512x512	0.0970 ms	0.0843 ms (15.1327%)	0.1004 ms (-3.3898%)
768x768	0.0941 ms	0.083 ms (13.27%)	0.0964 ms (-2.4113%)
1024x1024	0.0920 ms	0.0823 ms (11.7509%)	0.0953 ms (-3.4376%)
BVH PSR			
256x256	0.0698 ms	0.069 ms (1.1133%)	0.0726 ms (-3.8382%)
512x512	0.0664 ms	0.0662 ms (0.3718%)	0.0706 ms (-5.8874%)
768x768	0.0643 ms	0.0639 ms (0.566%)	0.0685 ms (-6.2105%)
1024x1024	0.063 ms	0.0625 ms (0.8186%)	0.0671 ms (-6.0543%)

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.13: BART Museum Average Traversal Times (Per Pixel)

Robots	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	0.0833 ms	0.073 ms (14.0007%)	0.086 ms (-3.223%)
512x512	0.072 ms	0.0626 ms (15.109%)	0.074 ms (-2.7105%)
768x768	0.0671 ms	0.0582 ms (15.4219%)	0.0688 ms (-2.42%)
1024x1024	0.0644 ms	0.0557 ms (15.6435%)	0.0659 ms (-2.2348%)
BVH P			
256x256	0.1071 ms	0.1069 ms (0.1449%)	0.1106 ms (-3.1847%)
512x512	0.0968 ms	0.0967 ms (0.1338%)	0.1006 ms (-3.7244%)
768x768	0.0934 ms	0.0933 ms (0.1051%)	0.0972 ms (-3.8301%)
1024x1024	0.0918 ms	0.0917 ms (0.0885%)	0.0955 ms (-3.8903%)
K-d PS			
256x256	0.1606 ms	0.1399 ms (14.7963%)	0.1658 ms (-3.1559%)
512x512	0.1405 ms	0.1215 ms (15.6348%)	0.1443 ms (-2.64%)
768x768	0.1316 ms	0.1136 ms (15.8755%)	0.1349 ms (-2.3813%)
1024x1024	0.1268 ms	0.1093 ms (16.0186%)	0.1297 ms (-2.2043%)
BVH PS			
256x256	0.2017 ms	0.2002 ms (0.771%)	0.2092 ms (-3.5812%)
512x512	0.1819 ms	0.1808 ms (0.6158%)	0.1892 ms (-3.8525%)
768x768	0.1751 ms	0.1741 ms (0.5633%)	0.1823 ms (-3.9296%)
1024x1024	0.1718 ms	0.1709 ms (0.5572%)	0.1789 ms (-3.952%)
K-d PSR			
256x256	0.1637 ms	0.1425 ms (14.8959%)	0.1693 ms (-3.2603%)
512x512	0.1439 ms	0.1242 ms (15.8977%)	0.148 ms (-2.7749%)
768x768	0.135 ms	0.1162 ms (16.2296%)	0.1385 ms (-2.4986%)
1024x1024	0.1301 ms	0.1117 ms (16.4519%)	0.1332 ms (-2.3097%)
BVH PSR			
256x256	0.2272 ms	0.2241 ms (1.405%)	0.2344 ms (-3.0385%)
512x512	0.199 ms	0.1965 ms (1.2761%)	0.2056 ms (-3.1678%)
768x768	0.1886 ms	0.1859 ms (1.4585%)	0.1944 ms (-3.0153%)
1024x1024	0.1834 ms	0.1803 ms (1.7432%)	0.1886 ms (-2.7648%)

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.14: BART Robots Average Traversal Times (Per Pixel)

5.2.3 MGF

With the MGF set of models increasing the size of the L1 cache leads to a just under 17% performance increase in certain cases.

The first scene we looked at was the conference scene, we can see that the BVH outperforms the K-d tree in all cases here.

The second scene is the office scene. The results of this experiment carried out on the office scene show that again the BVH is faster in all cases.

Lastly, the final scene is the theatre scene. In the case of this scene the K-d tree outperforms the BVH in all cases.

Conference	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	0.2325 ms	0.2081 ms (11.7267%)	0.2525 ms (-7.9313%)
512x512	0.1765 ms	0.1564 ms (12.8391%)	0.1898 ms (-7.0319%)
768x768	0.1499 ms	0.1322 ms (13.3959%)	0.1601 ms (-6.3577%)
1024x1024	0.134 ms	0.1178 ms (13.7478%)	0.1424 ms (-5.8917%)
BVH P			
256x256	0.1524 ms	0.1516 ms (0.5106%)	0.1568 ms (-2.811%)
512x512	0.1048 ms	0.1043 ms (0.4578%)	0.1082 ms (-3.1419%)
768x768	0.0894 ms	0.089 ms (0.4274%)	0.0925 ms (-3.3288%)
1024x1024	0.082 ms	0.0817 ms (0.3906%)	0.0849 ms (-3.4414%)
K-d PS			
256x256	0.5469 ms	0.4815 ms (13.5799%)	0.5929 ms (-7.7594%)
512x512	0.4202 ms	0.3685 ms (14.0479%)	0.4509 ms (-6.7921%)
768x768	0.3606 ms	0.3156 ms (14.2709%)	0.3842 ms (-6.1414%)
1024x1024	0.3249 ms	0.2837 ms (14.5048%)	0.3445 ms (-5.6903%)
BVH PS			
256x256	0.4632 ms	0.4574 ms (1.2845%)	0.4763 ms (-2.7457%)
512x512	0.3148 ms	0.3112 ms (1.167%)	0.3248 ms (-3.0522%)
768x768	0.2645 ms	0.2617 ms (1.0641%)	0.2734 ms (-3.2844%)
1024x1024	0.2394 ms	0.2371 ms (0.9617%)	0.248 ms (-3.4602%)
K-d PSR			
256x256	N/A	N/A	N/A
512x512	N/A	N/A	N/A
768x768	N/A	N/A	N/A
1024x1024	N/A	N/A	N/A
BVH PSR			
256x256	0.5527 ms	0.5452 ms (1.3804%)	0.5687 ms (-2.8125%)
512x512	0.3786 ms	0.3741 ms (1.2005%)	0.3898 ms (-2.8837%)
768x768	0.31 ms	0.3066 ms (1.1139%)	0.3199 ms (-3.0765%)
1024x1024	0.2746 ms	0.2718 ms (1.03%)	0.2837 ms (-3.226%)

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.15: MGF Conference Average Traversal Times (Per Pixel)

Office	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	0.121 ms	0.1085 ms (11.4605%)	0.1292 ms (-6.3926%)
512x512	0.0942 ms	0.0828 ms (13.7335%)	0.0996 ms (-5.4228%)
768x768	0.0827 ms	0.0722 ms (14.587%)	0.087 ms (-4.9436%)
1024x1024	0.0763 ms	0.0664 ms (15.0015%)	0.08 ms (-4.5908%)
BVH P			
256x256	0.0841 ms	0.084 ms (0.1313%)	0.087 ms (-3.2648%)
512x512	0.0648 ms	0.0646 ms (0.1938%)	0.0671 ms (-3.502%)
768x768	0.0579 ms	0.0578 ms (0.1803%)	0.0601 ms (-3.5941%)
1024x1024	0.0543 ms	0.0542 ms (0.172%)	0.0564 ms (-3.6463%)
K-d PS			
256x256	0.4737 ms	0.418 ms (13.3174%)	0.4999 ms (-5.2458%)
512x512	0.3785 ms	0.329 ms (15.0492%)	0.3965 ms (-4.5483%)
768x768	0.3388 ms	0.2927 ms (15.7376%)	0.3535 ms (-4.1584%)
1024x1024	0.3164 ms	0.2729 ms (15.9465%)	0.3291 ms (-3.8617%)
BVH PS			
256x256	0.3692 ms	0.3642 ms (1.384%)	0.3812 ms (-3.154%)
512x512	0.2878 ms	0.2845 ms (1.1684%)	0.2984 ms (-3.5305%)
768x768	0.2589 ms	0.2563 ms (1.0015%)	0.269 ms (-3.7686%)
1024x1024	0.244 ms	0.2418 ms (0.8946%)	0.2538 ms (-3.8668%)
K-d PSR			
256x256	0.4698 ms	0.4135 ms (13.6156%)	0.4959 ms (-5.2589%)
512x512	0.3763 ms	0.3258 ms (15.4893%)	0.3943 ms (-4.5666%)
768x768	0.3375 ms	0.2902 ms (16.313%)	0.3519 ms (-4.0771%)
1024x1024	0.3156 ms	0.2705 ms (16.6769%)	0.328 ms (-3.786%)
BVH PSR			
256x256	0.3784 ms	0.3738 ms (1.2163%)	0.391 ms (-3.2308%)
512x512	0.2928 ms	0.29 ms (0.9609%)	0.3037 ms (-3.5727%)
768x768	0.2626 ms	0.2605 ms (0.8046%)	0.2728 ms (-3.772%)
1024x1024	0.2469 ms	0.2453 ms (0.6702%)	0.2569 ms (-3.8867%)

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.16: MGF Office Average Traversal Times (Per Pixel)

Theatre	16KB/48KB	48KB/16KB	L1 Disabled
K-d P			
256x256	0.1759 ms	0.155 ms (13.465%)	0.1856 ms (-5.2263%)
512x512	0.1405 ms	0.1228 ms (14.3772%)	0.147 ms (-4.4511%)
768x768	0.126 ms	0.1098 ms (14.7674%)	0.1313 ms (-4.0506%)
1024x1024	0.118 ms	0.1026 ms (14.9823%)	0.1227 ms (-3.8112%)
BVH P			
256x256	0.1561 ms	0.1555 ms (0.4044%)	0.1611 ms (-3.0663%)
512x512	0.1331 ms	0.1327 ms (0.3332%)	0.1379 ms (-3.4768%)
768x768	0.125 ms	0.1246 ms (0.2822%)	0.1297 ms (-3.6123%)
1024x1024	0.1209 ms	0.1206 ms (0.2451%)	0.1255 ms (-3.6817%)
K-d PS			
256x256	0.4423 ms	0.3877 ms (14.0781%)	0.4646 ms (-4.7982%)
512x512	0.358 ms	0.3114 ms (14.9575%)	0.3734 ms (-4.1141%)
768x768	0.3232 ms	0.2802 ms (15.3497%)	0.3358 ms (-3.7541%)
1024x1024	0.3036 ms	0.2628 ms (15.5096%)	0.3147 ms (-3.5356%)
BVH PS			
256x256	0.4474 ms	0.4417 ms (1.2794%)	0.4616 ms (-3.0861%)
512x512	0.3669 ms	0.3629 ms (1.0909%)	0.38 ms (-3.4488%)
768x768	0.3374 ms	0.3341 ms (0.9734%)	0.3501 ms (-3.6239%)
1024x1024	0.3221 ms	0.3192 ms (0.9099%)	0.3345 ms (-3.7107%)
K-d PSR			
256x256	0.5395 ms	0.4782 ms (12.8177%)	0.5705 ms (-5.4453%)
512x512	0.4518 ms	0.3978 ms (13.5715%)	0.4747 ms (-4.8339%)
768x768	0.4135 ms	0.3624 ms (14.1061%)	0.4329 ms (-4.4901%)
1024x1024	0.3909 ms	0.3416 ms (14.4154%)	0.4083 ms (-4.28%)
BVH PSR			
256x256	0.9865 ms	0.9725 ms (1.438%)	1.0109 ms (-2.4164%)
512x512	0.7655 ms	0.7555 ms (1.3289%)	0.7854 ms (-2.5286%)
768x768	0.6672 ms	0.6585 ms (1.3182%)	0.6851 ms (-2.6222%)
1024x1024	0.6096 ms	0.6017 ms (1.3071%)	0.6265 ms (-2.7028%)

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.17: MGF Theatre Average Traversal Times (Per Pixel)

5.2.4 Dynamic Scene Traversal

In the case of the dynamic scene the K-d tree had faster traversal times in all cases, but this does not mean that the K-d tree is better to use for a dynamic scene. When looking at a dynamic scene we must also consider the cost of rebuilding or refitting the acceleration data structure, although this is outside the scope of this dissertation. Two papers that look specifically at the construction of these acceleration data structures are "Fast BVH Construction on GPUs" [18] and "Real-Time KD-Tree Construction on Graphics Hardware" [37]. Aside from the actual build time another important factor is the quality of the data structure which will effect the traversal.

Dynamic	K-d Tree	BVH
256x256	1.091703440 ms	2.450178239 ms
512x512	1.315079226 ms	2.448173646 ms
768x768	1.340206107 ms	1.949641679 ms
1024x1024	1.424558189 ms	1.827218403 ms

Table 5.18: Dynamic Scene Primary Ray Average Traversal Times

Dynamic	K-d Tree	BVH
256x256	1.639874173 ms	4.622630716 ms
512x512	2.291114947 ms	4.700892774 ms
768x768	2.236793679 ms	3.640467704 ms
1024x1024	2.336598239 ms	3.435019687 ms

Table 5.19: Dynamic Scene Primary & Shadow Rays Average Traversal Times (Per-Pixel)

5.3 Cache Performance

In this experiment we looked at the cache performance of the two acceleration data structures. To achieve this I looked at the cache hit rate for both L1 and L2 caches, I then increased the size of the L1 cache to see the effect on the cache hit rates. In this particular implementation I found that the cache performance of K-d trees was higher than that of BVHs. I also tested this experiment on a Kepler based GPU and found that the L1 cache hit rate were zero percent, this is because with the Kepler

architecture the L1 cache is reserved only for local memory accesses such as register spills and stack data while the L2 cache is used for global loads. [25]

5.3.1 Stanford

When carrying out this experiment using the Stanford models the K-d tree exhibits a very high hit rate while the hit rate of the BVH is relatively low, more than 50% lower in cases. The results of this experiment can be seen in tables 5.20 and 5.21 below.

5.3.2 BART

In the case of both the K-d tree and BVH exhibit a very high cache hit rate in the majority of cases. Increasing the size of the L1 cache increases the L1 cache hit rate in all cases and in some cases with the BVH there is around a 13% increase in the hit rate. Overall the K-d tree hit rate is higher than the BVH by around 5% to 10% in most cases. The results of this experiment can be seen in the tables 5.22, 5.23 and 5.24 below.

5.3.3 MGF

The last set of models we looked at the cache performance of were the MGF models. Again this shows similar results to the BART models where both acceleration data structures have high cache hit rates with the K-d tree being slightly higher. The results of this experiment can be seen in the tables 5.25, 5.26 and 5.27 below.

	L1	**L2**	**L1 (Prefer L1)**	**L2 (Prefer L1)**
K-d Bunny P				
256x256	88.7254%	71.7455%	91.9615%	74.1596%
512x512	94.2964%	94.2964%	95.8374%	85.7804%
768x768	96.2554%	96.2554%	96.9571%	89.1308%
1024x1024	97.0987%	97.0987%	97.4707%	90.7466%
BVH Bunny P				
256x256	41.9802%	48.3771%	47.4030%	44.8126%
512x512	51.7407%	73.0808%	58.5077%	69.7721%
768x768	59.1198%	81.3547%	66.2590%	78.3362%
1024x1024	64.4346%	85.9042%	71.7949%	83.1064%
K-d Dragon P				
256x256	81.94%	81.94%	84.7443%	43.781%
512x512	87.8225%	87.8225%	91.1692%	67.1093%
768x768	91.6905%	91.6905%	94.0944%	78.9125%
1024x1024	93.9338%	93.9338%	95.4931%	84.1211%
BVH Dragon P				
256x256	31.0469%	8.9269%	34.1189%	6.4467%
512x512	37.1026%	21.5203%	42.0901%	17.4883%
768x768	40.9344%	36.3629%	47.2147%	31.3302%
1024x1024	44.3192%	49.8583%	51.4148%	44.6086%
K-d Buddha P				
256x256	81.4238%	35.9401%	83.753%	39.6696%
512x512	86.4045%	53.6828%	89.5807%	60.3979%
768x768	90.1743%	68.7898%	92.8409%	73.3896%
1024x1024	92.6473%	78.9089%	94.5553%	80.242%
BVH Buddha P				
256x256	30.3565%	7.4372%	32.9112%	5.286%
512x512	36.0842%	17.9975%	40.4944%	14.3361%
768x768	40.3394%	31.0218%	46.1537%	26.191%
1024x1024	43.8036%	43.6431%	50.5967%	38.3157%

P Primary Rays

Prefer L1 48KB L1 with 16KB Shared Memory

Table 5.20: Stanford K-d Primary Ray Cache Hit Rates

	L1	L2	L1 (Prefer L1)	L2 (Prefer L1)
K-d Bunny P				
256x256	89.0083%	71.8903%	91.8062%	72.1191%
512x512	94.373%	88.0727%	95.42%	83.0812%
768x768	96.1501%	91.8008%	96.5701%	86.7545%
1024x1024	96.8642%	93.3026%	97.0813%	88.3513%
BVH Bunny P				
256x256	42.1724%	66.8537%	54.7251%	62.3881%
512x512	53.7336%	85.5344%	67.7834%	81.1074%
768x768	61.1924%	90.2891%	74.1427%	86.7543%
1024x1024	66.5673%	92.4605%	78.0122%	89.5525%
K-d Dragon P				
256x256	81.6642%	35.1682%	83.762%	39.2567%
512x512	87.59%	57.3134%	90.0499%	62.4169%
768x768	91.5512%	74.0978%	93.1141%	73.7321%
1024x1024	91.5535%	74.1044%	94.5708%	78.3563%
BVH Dragon P				
256x256	28.6822%	12.3618%	33.4067%	9.5204%
512x512	34.521%	25.407%	42.6423%	21.1018%
768x768	39.2113%	41.3351%	50.0489%	35.7574%
1024x1024	43.1911%	56.4027%	55.5731%	50.1062%
K-d Buddha P				
256x256	80.6927%	29.7373%	82.3235%	32.5334%
512x512	84.8232%	45.186%	87.5864%	51.0954%
768x768	88.2942%	60.1196%	90.958%	65.2007%
1024x1024	90.8436%	71.8014%	92.9431%	73.292%
BVH Buddha P				
256x256	27.2272%	10.1599%	31.0499%	7.6032%
512x512	31.8826%	19.0275%	38.1697%	15.2827%
768x768	36.1696%	29.7772%	44.8475%	24.8189%
1024x1024	39.6891%	41.0252%	50.0185%	35.2469%

P Primary Rays

Prefer L1 48KB L1 with 16KB Shared Memory

Table 5.21: Stanford K-d Primary & Shadow Ray Cache Hit Rates

	L1	**L2**	**L1 (Prefer L1)**	**L2 (Prefer L1)**
K-d P				
256x256	97.7413%	69.79%	97.9075%	67.2745%
512x512	98.6715%	64.3775%	98.8055%	60.866%
768x768	99.0783%	59.1768%	99.164%	55.6125%
1024x1024	99.286%	55.451%	99.3558%	51.4198%
BVH P				
256x256	91.5188%	65.0053%	93.8635%	60.9183%
512x512	95.5228%	57.9355%	97.4063%	52.0188%
768x768	96.952%	52.2713%	98.4738%	45.5565%
1024x1024	97.668%	48.0945%	98.9575%	40.6923%
K-d PS				
256x256	98.3215%	81.7288%	98.7145%	78.1905%
512x512	98.9815%	77.0075%	99.2623%	73.379%
768x768	99.2893%	73.045%	99.4865%	69.3603%
1024x1024	99.446%	69.905%	99.614%	65.9145%
BVH PS				
256x256	76.1943%	56.7783%	90.3443%	64.126%
512x512	77.5993%	52.9188%	92.0555%	56.1723%
768x768	71.4133%	52.8798%	89.8353%	52.2493%
1024x1024	71.6673%	52.0098%	90.1033%	49.9428%
K-d PSR				
256x256	98.072%	80.6815%	98.6275%	78.689%
512x512	98.6533%	76.6678%	99.1658%	74.1268%
768x768	98.9973%	73.21%	99.3918%	70.573%
1024x1024	99.1845%	70.659%	99.5178%	67.6028%
BVH PSR				
256x256	88.9655%	64.3745%	95.0988%	65.7275%
512x512	89.0273%	61.6588%	95.4888%	58.334%
768x768	68.0555%	54.339%	86.2805%	53.4748%
1024x1024	68.0825%	53.823%	86.6735%	52.4505%

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.22: BART Robots Cache Hit Rates

	L1	L2	L1 (Prefer L1)	L2 (Prefer L1)
K-d P				
256x256	96.8963%	79.0955%	97.2358%	73.6788%
512x512	98.074%	80.427%	98.2738%	74.0298%
768x768	98.5803%	78.5195%	98.7253%	71.718%
1024x1024	98.8523%	75.6958%	98.9805%	68.78%
BVH P				
256x256	86.4695%	63.5003%	89.7248%	58.3555%
512x512	92.7368%	68.079%	95.0098%	62.3025%
768x768	95.0435%	66.8953%	96.871%	60.191%
1024x1024	96.227%	64.4285%	97.761%	56.8758%
K-d PS				
256x256	96.349%	90.207%	97.746%	89.1208%
512x512	97.8343%	93.4165%	98.5883%	90.9583%
768x768	98.4113%	93.6118%	98.9255%	90.988%
1024x1024	98.7%	93.1858%	99.1125%	90.3358%
BVH PS				
256x256	77.913%	68.1305%	88.3568%	67.4905%
512x512	79.431%	69.0908%	90.652%	72.0895%
768x768	74.8335%	67.0318%	89.1448%	67.1693%
1024x1024	75.4248%	65.5963%	89.7435%	66.245%
K-d PSR				
256x256	94.6063%	83.1018%	97.0385%	83.7025%
512x512	95.9943%	85.6793%	97.8038%	85.0753%
768x768	96.6113%	86.4218%	98.1223%	85.4313%
1024x1024	96.991%	86.7428%	98.3088%	85.5685%
BVH PSR				
256x256	84.6468%	63.944%	90.3708%	60.2533%
512x512	84.821%	64.6758%	91.0345%	61.453%
768x768	71.955%	60.2195%	83.9625%	55.8888%
1024x1024	72.0948%	60.4005%	84.6515%	56.2023%

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.23: BART Kitchen Cache Hit Rates

	L1	**L2**	**L1 (Prefer L1)**	**L2 (Prefer L1)**
K-d P				
256x256	97.7113%	88.2573%	97.9967%	83.7513%
512x512	98.6953%	87.95%	98.8073%	82.2633%
768x768	99.0113%	85.438%	99.088%	79.036%
1024x1024	99.1767%	82.78%	99.262%	75.4333%
BVH P				
256x256	85.9273%	71.3567%	89.3173%	67.896%
512x512	92.408%	71.014%	94.808%	66.894%
768x768	94.6433%	68.0147%	96.8%	62.6333%
1024x1024	95.8167%	64.9453%	97.784%	58.478%
K-d PS				
256x256	97.462%	94.6767%	98.406%	92.5467%
512x512	98.576%	95.7227%	99.024%	93.354%
768x768	98.9207%	95.216%	99.262%	92.33%
1024x1024	99.098%	94.4973%	99.3993%	90.868%
BVH PS				
256x256	72.7587%	67.2787%	85.3713%	74.1453%
512x512	73.394%	63.9453%	87.728%	75.1167%
768x768	67.87%	60.046%	86.1973%	69.282%
1024x1024	67.9747%	58.6933%	86.642%	68.2953%
K-d PSR				
256x256	95.2353%	92.054%	97.5273%	91.732%
512x512	96.6413%	93.742%	98.2687%	92.994%
768x768	97.298%	94.2993%	98.58%	93.072%
1024x1024	97.6967%	94.622%	98.7647%	93.246%
BVH PSR				
256x256	86.7153%	73.2727%	91.8567%	76.646%
512x512	85.6567%	70.706%	92.0467%	76.1287%
768x768	65.322%	60.8047%	81.3373%	65.6427%
1024x1024	65.084%	60.1087%	81.878%	65.496%

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.24: BART Museum Cache Hit Rates

	L1	L2	L1 (Prefer L1)	L2 (Prefer L1)
K-d P				
256x256	96.3263%	79.807%	97.522%	80.7903%
512x512	97.4533%	86.9513%	98.404%	86.2403%
768x768	97.9778%	89.4555%	98.761%	86.929%
1024x1024	98.2965%	90.0003%	98.9623%	86.6288%
BVH P				
256x256	76.069%	62.9155%	81.8743%	58.0993%
512x512	84.885%	73.2975%	89.9703%	67.927%
768x768	88.6505%	76.1685%	93.0978%	70.566%
1024x1024	90.725%	76.502%	94.7595%	70.3408%
K-d PS				
256x256	95.6685%	78.8983%	97.869%	84.3118%
512x512	96.9843%	88.2203%	98.64%	90.6388%
768x768	97.6148%	91.8575%	98.9423%	92.1405%
1024x1024	97.9815%	93.1853%	99.1023%	92.6218%
BVH PS				
256x256	73.3758%	57.5068%	82.5275%	51.4895%
512x512	74.126%	63.997%	84.9703%	62.5568%
768x768	69.7213%	61.9723%	85.9915%	66.4673%
1024x1024	69.7008%	61.326%	86.5578%	67.7203%
K-d PSR				
256x256	N/A	N/A	N/A	N/A
512x512	N/A	N/A	N/A	N/A
768x768	N/A	N/A	N/A	N/A
1024x1024	N/A	N/A	N/A	N/A
BVH PSR				
256x256	86.3325%	59.4543%	79.561%	49.9278%
512x512	85.7995%	64.64%	81.4608%	57.4695%
768x768	67.0698%	59.3788%	82.584%	61.539%
1024x1024	66.8613%	59.3833%	83.3263%	62.4985%

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.25: MGF Conference Cache Hit Rates

	L1	L2	L1 (Prefer L1)	L2 (Prefer L1)
K-d P				
256x256	97.678%	82.98%	97.916%	78.98%
512x512	98.432%	82.796%	98.638%	76.464%
768x768	98.856%	80.106%	99.016%	73.636%
1024x1024	99.106%	77.054%	99.236%	70.132%
BVH P				
256x256	88.562%	70.758%	90.762%	67.294%
512x512	93.892%	68.878%	95.522%	64.848%
768x768	96.062%	65.38%	97.356%	59.71%
1024x1024	97.06%	61.572%	98.224%	54.882%
K-d PS				
256x256	97.684%	94.582%	98.69%	92.106%
512x512	98.586%	95.638%	99.208%	93.416%
768x768	98.98%	95.494%	99.442%	93.276%
1024x1024	99.162%	94.728%	99.572%	92.134%
BVH PS				
256x256	75.244%	69.666%	85.712%	70.698%
512x512	75.93%	66.984%	87.32%	70.704%
768x768	70.644%	63.624%	87.924%	70.072%
1024x1024	70.744%	62.286%	88.272%	68.978%
K-d PSR				
256x256	97.69%	94.476%	98.724%	93.022%
512x512	98.584%	95.096%	99.218%	93.87%
768x768	98.944%	94.644%	99.446%	93.242%
1024x1024	99.136%	94.136%	99.57%	92.404%
BVH PSR				
256x256	83.286%	69.394%	82.646%	66.06%
512x512	83.182%	66.762%	84.152%	65.606%
768x768	68.014%	60.802%	84.796%	64.274%
1024x1024	67.924%	59.714%	85.258%	63.526%

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.26: MGF Office Cache Hit Rates

	L1	**L2**	**L1 (Prefer L1)**	**L2 (Prefer L1)**
K-d P				
256x256	98.0003%	82.9588%	98.1758%	78.3093%
512x512	98.7038%	79.3348%	98.8655%	74.9638%
768x768	99.0235%	75.9298%	99.183%	71.515%
1024x1024	99.2113%	72.7668%	99.3693%	68.4588%
BVH P				
256x256	86.5248%	79.116%	91.4868%	75.1303%
512x512	92.1983%	75.623%	94.9525%	69.8095%
768x768	94.4308%	71.753%	97.79%	64.4143%
1024x1024	95.5773%	68.4743%	98.4958%	59.876%
K-d PS				
256x256	97.9488%	93.926%	98.7278%	91.997%
512x512	98.7753%	94.3013%	99.2185%	91.9113%
768x768	99.1065%	93.3765%	99.4435%	90.8473%
1024x1024	99.2865%	92.2063%	99.572%	89.5775%
BVH PS				
256x256	77.2243%	72.756%	90.2973%	77.6628%
512x512	78.6733%	68.9773%	92.504%	77.8218%
768x768	74.6138%	65.2498%	90.32%	69.4663%
1024x1024	74.9068%	63.7095%	90.7123%	67.89%
K-d PSR				
256x256	96.4188%	89.49%	98.103%	89.1083%
512x512	97.015%	90.2285%	98.478%	89.925%
768x768	97.3238%	90.3758%	98.6543%	89.764%
1024x1024	97.5343%	90.2555%	98.767%	89.5728%
BVH PSR				
256x256	87.6438%	73.6593%	93.1338%	73.9283%
512x512	87.8355%	73.3208%	93.6445%	75.108%
768x768	71.4448%	65.3403%	84.0355%	65.3415%
1024x1024	71.5355%	64.6305%	84.4715%	65.0268%

P Primary Rays

PS Primary & Shadow Rays

PSR Primary, Shadow & Reflection Rays

Table 5.27: MGF Theatre Cache Hit Rates

5.3.4 Dynamic Scene

This experiment looks at the cache performance of the dynamic scene. The results of this can be seen in table 5.28 where the K-d tree exhibts much higher cache hit rates for both L1 and L2 caches compared to the BVH.

Dynamic Scene	**L1**	**L2**
K-d P		
256x256	80.7616%	31.4108%
512x512	84.3577%	46.2912%
768x768	87.3612%	55.8229%
1024x1024	89.6358%	63.6617%
BVH P		
256x256	29.2111%	7.3424%
512x512	33.4784%	15.7988%
768x768	37.8214%	22.5094%
1024x1024	41.8186%	28.5494%
K-d PS		
256x256	79.3588%	27.4060%
512x512	82.7308%	41.4814%
768x768	85.9977%	51.3539%
1024x1024	88.7248%	59.6181%
BVH PS		
256x256	27.1543%	9.3232%
512x512	31.3039%	18.9633%
768x768	35.8362%	26.2707%
1024x1024	40.2724%	32.1637%

P Primary Rays

PS Primary & Shadow Rays

Table 5.28: Dynamic Scene Cache Hit Rates

5.4 Branch Divergence

In this section the results of the branch divergence experiment are presented. Branch divergence is important as it can have a direct effect on the performance of a piece of code due to the SM units not being fully utilised. Looking at these results of this experiment we can see that for BVHs the branch divergence percentage is very low, while with the K-d tree it sits around ten to thirteen percent. This means that the K-d tree may be able to be optimised to gain better performance from a branch divergence perspective as the warp is not fully utilised when branch divergence occurs.

5.4.1 Stanford

The results for the branch divergence experiment when using the Stanford models should a divergent branch rate of around 13% for the K-d tree. In the case of the BVH the divergent branch rates are very low, less than 1% in all cases. The results of this experiment can be seen in the tables 5.29 and 5.30 below.

Bunny	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	12.3271%	11.5262%	10.7446%	10.0655%
Shadow	12.2952%	11.3501%	10.5450%	9.82812%
BVH				
Primary	0.16306%	0.29513%	0.37713%	0.43166%
Shadow	0.09765%	0.17580%	0.22524%	0.25670%

Table 5.29: Stanford Bunny Branch Divergence

Dragon	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	13.0173%	12.7861%	12.7861%	11.9662%
Shadow	13.1575%	12.9589%	12.9589%	12.1049%
BVH				
Primary	0.04943%	0.09297%	0.09297%	0.18208%
Shadow	0.02675%	0.05204%	0.05204%	0.10196%

Table 5.30: Stanford Dragon Branch Divergence

Buddha	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	12.8408%	12.7178%	12.4213%	12.0739%
Shadow	12.9624%	12.8977%	12.5958%	12.2889%
BVH				
Primary	0.05601%	0.10336%	0.07722%	0.20588%
Shadow	0.02827%	0.05184%	0.15518%	0.10252%

Table 5.31: Stanford Buddha Branch Divergence

5.4.2 MGF

With the MGF models we can see that the K-d tree exhibits a branch divergence rate of 3% to 7% while again the BVH has a branch divergence rate of less than one percent. We can see the results of this experiment in the tables 5.32, 5.33 and 5.34 below.

Theatre	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	6.5332%	4.8082%	3.8410%	3.2119%
Shadow	6.7160%	4.9502%	3.9702%	3.3312%
Reflect	7.2184%	6.0835%	5.3738%	4.8726%
BVH				
Primary	0.1113%	0.1118%	0.1110%	0.1102%
Shadow	0.0916%	0.1038%	0.1092%	0.1123%
Reflect	0.0643%	0.0725%	0.0777%	0.0815%

Table 5.32: MGF Theatre Branch Divergence

Conference	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	7.5441%	6.5076%	5.7711%	5.1906%
Shadow	7.0218%	5.9486%	5.2271%	4.6778%
Reflect	N/A	N/A	N/A	N/A
BVH				
Primary	0.1545%	0.1699%	0.1746%	0.1771%
Shadow	0.1046%	0.1345%	0.1500%	0.1597%
Reflect	0.0958%	0.1222%	0.1378%	0.1483%

Table 5.33: MGF Conference Branch Divergence

Office	256x256	512x512	768x768	1024x1024
K-d Tree				
Primary	6.6141%	4.9841%	4.0503%	3.4318%
Shadow	6.7019%	4.9063%	3.9104%	3.2688%
Reflect	6.7083%	4.9313%	3.9353%	3.2932%
BVH				
Primary	0.2022%	0.2229%	0.2324%	0.2388%
Shadow	0.1481%	0.1779%	0.1916%	0.1996%
Reflect	0.1477%	0.1772%	0.1909%	0.1989%

Table 5.34: MGF Office Branch Divergence

5.4.3 BART

When carrying out this experiment with the BART set of models we get a branch divergence rate between 2% to 5% with the K-d tree and again the BVH divergent branch rate is less than one percent. We can see the results of this experiment in the tables 5.35, 5.36 and 5.37 below.

Robots	256x256	512x512	768x768	1024x1024
K-d Tree				
Primary	4.9341%	3.3518%	2.5719%	2.1057%
Shadow	4.8862%	3.3007%	2.5282%	2.0733%
Reflect	5.0801%	3.5670%	2.7996%	2.3371%
BVH				
Primary	0.1087%	0.1173%	0.1204%	0.1219%
Shadow	0.1113%	0.1206%	0.1240%	0.1257%
Reflect	0.1034%	0.1147%	0.1193%	0.1217%

Table 5.35: BART Robots Branch Divergence

Kitchen	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	8.1629%	6.3662%	5.2883%	4.5539%
Shadow	8.4829%	6.6189%	5.5156%	4.7678%
Reflect	9.3218%	7.9368%	7.0375%	6.3906%
BVH				
Primary	0.1293%	0.1322%	0.1321%	0.1319%
Shadow	0.0905%	0.1114%	0.1203%	0.1253%
Reflect	0.0678%	0.0831%	0.0920%	0.0982%

Table 5.36: BART Kitchen Branch Divergence

Museum	**256x256**	**512x512**	**768x768**	**1024x1024**
K-d Tree				
Primary	9.5702%	7.6559%	6.4748%	5.6434%
Shadow	9.2671%	7.3506%	6.1856%	5.3772%
Reflect	10.4329%	9.1405%	8.2340%	7.5476%
BVH				
Primary	0.1344%	0.1606%	0.1703%	0.1752%
Shadow	0.1416%	0.1747%	0.1881%	0.1952%
Reflect	0.0878%	0.1183%	0.1364%	0.1484%

Table 5.37: BART Museum Branch Divergence

5.5 Instruction Statistics

The results of the instruction statistics experiment are presented in this section. The experiment was carried out over four hundred kernel launches where each kernel launch renders a single frame.

5.5.1 Stanford

The results of the instruction statistics experiment executed with the Stanford models shows that there is some correlation to the results of the traversal experiment where as the resolution increases the gap between the K-d tree and BV narrows. The results of this experiment can be seen in the tables 5.38, 5.39 and 5.40 below.

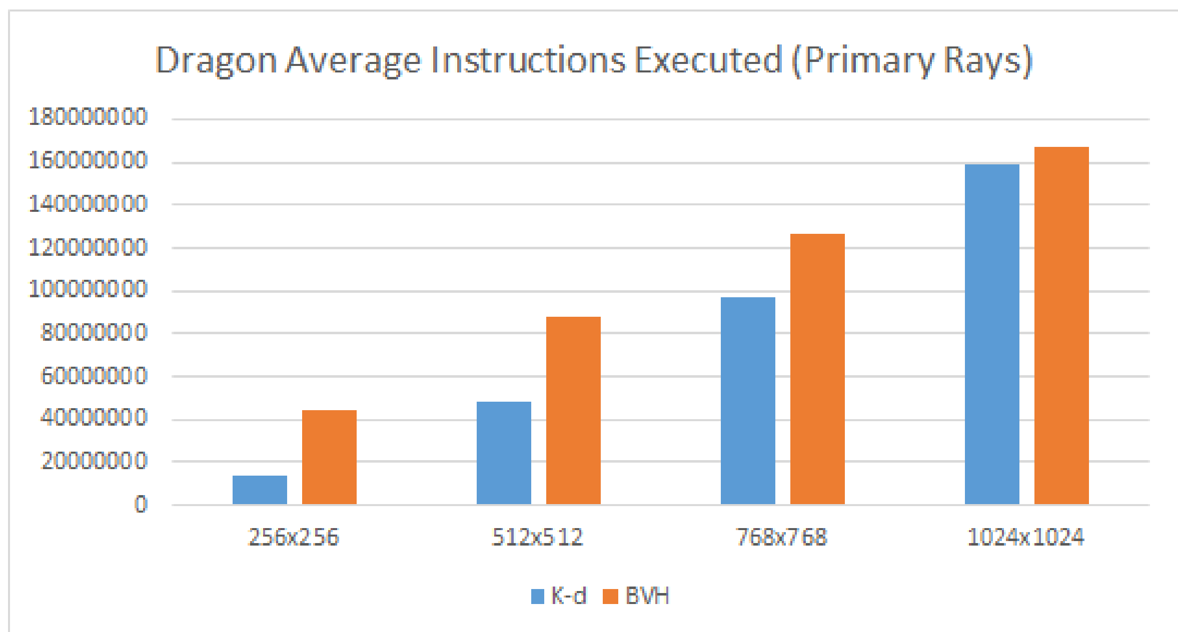


Figure 5.1: Dragon Primary Rays Instructions Executed

5.5.2 BART

The instruction statistics experiment using the BART set of models shows that the BVH executes more instructions during the robots and kitchen scenes which correlates

Bunny	Min	Max	Avg
Primary K-d			
256x256	8,113,908	13,821,294	10,689,697
512x512	25,789,073	44,737,749	34,735,123
768x768	50,421,151	88,162,328	68,504,532
1024x1024	81,381,976	141,499,549	110,479,822
Primary BVH			
256x256	10,259,705	16,944,892	13,146,725
512x512	21,166,874	32,657,490	26,101,344
768x768	35,662,501	53,705,256	43,527,602
1024x1024	53,865,371	80,495,600	65,599,326
Primary, Shadow K-d			
256x256	16,603,083	26,685,409	21,306,060
512x512	52,240,964	87,138,315	68,235,223
768x768	102,646,942	173,398,160	134,479,161
1024x1024	165,052,302	280,365,628	216,162,378
Primary, Shadow BVH			
256x256	22,040,755	28,567,205	24,760,668
512x512	45,116,619	57,611,733	50,428,857
768x768	75,198,461	95,232,629	84,242,856
1024x1024	113,533,466	143,505,662	127,528,932

Table 5.38: Instructions Executed Stanford Bunny

Dragon	**Min**	**Max**	**Avg**
Primary K-d			
256x256	9,810,782	19,040,104	14,152,982
512x512	33,393,987	64,770,057	47,888,486
768x768	68,016,269	132,084,320	96,986,543
1024x1024	111,611,486	217,125,478	159,368,537
Primary BVH			
256x256	23,537,422	68,516,898	44,444,564
512x512	46,063,384	125,441,885	88,310,362
768x768	68,928,640	174,121,535	126,357,093
1024x1024	94,100,223	228,627,963	166,748,826
Primary, Shadow K-d			
256x256	23,370,475	37,054,622	29,147,271
512x512	78,823,337	126,290,561	98,050,145
768x768	159,499,607	256,865,496	198,245,056
1024x1024	260,283,086	421,853,659	323,800,439
Primary, Shadow BVH			
256x256	74,650,892	106,911,410	90,992,207
512x512	153,367,748	210,955,588	185,736,296
768x768	225,309,683	308,279,530	271,221,728
1024x1024	300,313,293	406,280,033	357,970,041

Table 5.39: Instructions Executed Stanford Dragon

Buddha	**Min**	**Max**	**Avg**
Primary K-d			
256x256	5,485,062	13,391,526	10,362,370
512x512	18,460,528	46,263,618	35,245,910
768x768	37,310,467	94,356,821	71,562,185
1024x1024	61,392,361	155,703,732	117,758,349
Primary BVH			
256x256	13,653,712	52,337,691	36,253,867
512x512	31,045,644	105,081,374	76,185,802
768x768	47,004,872	153,494,709	111,550,610
1024x1024	64,008,190	197,192,819	146,123,606
Primary, Shadow K-d			
256x256	13,785,464	25,987,887	21,220,451
512x512	46,832,152	88,817,687	72,286,267
768x768	95,049,723	181,681,576	146,776,536
1024x1024	156,137,620	298,500,257	241,167,324
Primary, Shadow BVH			
256x256	58,262,629	92,241,679	77,329,485
512x512	126,406,565	196,892,463	170,539,690
768x768	192,863,632	291,627,325	257,035,295
1024x1024	255,670,909	380,905,597	338,346,077

Table 5.40: Instructions Executed Stanford Buddha

directly with the results of the full render experiment. The results of this experiment can be seen in the tables 5.41, 5.42 and 5.43 below.

5.5.3 MGF

When carrying out this experiment using the MGF models we can see that in the conference and office scenes the BVH executes less instructions than the K-d tree which shows some correlation to the full render experiment. The results of this experiment can be seen in the tables 5.44, 5.45 and 5.46 below.

Robots	**Min**	**Max**	**Avg**
Primary K-d			
256x256	3,893,960	19,080,977	9,703,300
512x512	15,436,000	61,200,474	33,421,200
768x768	34,352,243	121,605,285	70,027,189
1024x1024	60,657,049	198,705,107	119,394,738
Primary BVH			
256x256	6,928,423	21,036,077	14,501,004
512x512	27,289,497	73,557,999	52,795,158
768x768	61,298,382	157,952,756	114,987,234
1024x1024	108,543,525	274,514,668	201,111,415
Primary, Shadow K-d			
256x256	9,008,077	34,362,925	18,637,198
512x512	35,716,748	109,945,633	64,960,026
768x768	80,134,883	218,138,939	136,899,949
1024x1024	142,235,943	357,383,559	234,312,674
Primary, Shadow BVH			
256x256	12,517,192	39,412,250	26,893,168
512x512	49,379,705	135,818,771	97,475,622
768x768	110,802,706	289,572,993	211,809,459
1024x1024	196,366,131	501,104,144	370,034,776
Primary, Shadow, Reflect K-d			
256x256	8,993,357	40,151,500	19,969,385
512x512	35,670,317	126,904,435	68,876,338
768x768	80,039,650	249,282,651	144,230,498
1024x1024	142,075,106	405,586,261	245,617,725
Primary, Shadow, Reflect BVH			
256x256	11,546,323	53,537,135	28,094,209
512x512	45,554,743	165,353,722	97,877,201
768x768	102,217,739	333,908,341	208,766,957
1024x1024	181,157,263	562,925,193	360,683,141

Table 5.41: Instructions Executed BART Robots

Museum	**Min**	**Max**	**Avg**
Primary K-d			
256x256	6,915,683	24,174,990	18,393,939
512x512	24,120,852	76,965,655	59,430,693
768x768	51,630,527	151,682,443	119,131,803
1024x1024	89,252,944	248,263,007	196,681,745
Primary BVH			
256x256	6,003,565	15,205,904	12,254,406
512x512	23,184,787	47,364,325	38,978,725
768x768	51,212,868	99,777,941	81,240,263
1024x1024	87,620,119	171,853,325	139,200,067
Primary, Shadow K-d			
256x256	23,775,747	56,265,591	46,495,781
512x512	85,523,712	180,219,941	151,768,362
768x768	184,265,095	358,144,079	306,388,323
1024x1024	319,439,432	587,407,925	507,895,994
Primary, Shadow BVH			
256x256	18,012,363	40,359,200	33,511,315
512x512	67,244,864	121,298,101	103,876,158
768x768	148,006,798	245,764,159	213,406,456
1024x1024	260,140,661	414,139,306	362,306,073
Primary, Shadow, Reflect K-d			
256x256	27,909,854	111,816,926	89,539,788
512x512	98,005,465	358,388,986	290,216,904
768x768	208,265,363	710,647,850	580,092,571
1024x1024	358,790,136	1,159,457,233	952,226,235
Primary, Shadow, Reflect BVH			
256x256	20,738,613	123,330,855	91,935,270
512x512	74,346,218	331,648,786	251,475,802
768x768	161,265,755	611,743,865	471,190,428
1024x1024	281,492,451	965,165,374	751,378,516

Table 5.42: Instructions Executed BART Museum

Kitchen	Min	Max	Avg
Primary K-d			
256x256	2,202,990	21,504,402	14,480,209
512x512	8,735,391	67,863,897	46,618,121
768x768	19,598,835	134,604,958	93,636,909
1024x1024	34,787,269	219,757,721	154,656,529
Primary BVH			
256x256	7,742,298	26,768,593	18,133,653
512x512	30,224,992	75,639,120	57,285,970
768x768	67,526,863	149,275,073	118,577,499
1024x1024	119,647,759	248,506,499	202,204,437
Primary, Shadow K-d			
256x256	21,374,975	100,733,199	66,963,934
512x512	81,968,914	320,082,911	219,009,182
768x768	181,525,823	634,199,828	443,521,130
1024x1024	320,188,057	1,035,395,153	736,988,233
Primary, Shadow BVH			
256x256	41,898,795	123,177,816	87,539,293
512x512	164,097,920	345,722,982	266,108,855
768x768	366,810,402	671,126,230	539,373,130
1024x1024	649,838,268	1,103,591,463	907,716,152
Primary, Shadow, Reflect K-d			
256x256	21,075,353	151,920,019	94,517,448
512x512	80,880,321	492,378,465	308,438,507
768x768	179,166,880	986,674,577	621,356,706
1024x1024	316,075,336	1,614,622,767	1026,222,116
Primary, Shadow, Reflect BVH			
256x256	41,118,480	349,917,717	167,763,894
512x512	161,065,481	971,043,595	488,551,935
768x768	360,043,993	1,809,405,642	941,268,890
1024x1024	637,854,153	2,842,246,878	1,518,959,409

Table 5.43: Instructions Executed BART Kitchen

Conference	**Min**	**Max**	**Avg**
Primary K-d			
256x256	16,854,794	37,494,106	27,911,959
512x512	53,122,044	114,483,586	85,453,180
768x768	105,518,814	218,205,088	163,435,149
1024x1024	172,617,314	344,102,462	259,374,342
Primary BVH			
256x256	13,285,274	25,157,138	19,671,433
512x512	39,981,855	67,630,919	55,049,208
768x768	80,868,911	127,675,202	106,877,915
1024x1024	135,700,207	206,863,511	175,309,331
Primary, Shadow K-d			
256x256	38,689,920	79,027,809	64,141,677
512x512	123,778,986	245,734,974	199,838,447
768x768	247,125,460	474,459,892	386,564,576
1024x1024	407,126,661	754,599,232	618,477,868
Primary, Shadow BVH			
256x256	35,772,327	70,281,563	56,185,600
512x512	107,840,494	189,138,522	155,924,804
768x768	216,080,068	357,900,168	298,931,376
1024x1024	361,386,129	576,356,466	485,383,263
Primary, Shadow, Reflect K-d			
256x256	N/A	N/A	N/A
512x512	N/A	N/A	N/A
768x768	N/A	N/A	N/A
1024x1024	N/A	N/A	N/A
Primary, Shadow, Reflect BVH			
256x256	37,334,609	81,297,261	64,889,040
512x512	110,037,241	219,665,775	177,859,152
768x768	217,755,856	402,301,337	332,412,734
1024x1024	360,133,816	631,253,217	528,070,215

Table 5.44: Instructions Executed MGF Conference

Office	Min	Max	Avg
Primary K-d			
256x256	11,881,185	17,347,495	14,931,366
512x512	37,508,613	52,579,038	45,832,265
768x768	74,547,840	102,349,813	90,078,543
1024x1024	122,598,058	166,143,944	147,253,403
Primary BVH			
256x256	9,411,016	12,527,982	11,188,651
512x512	29,865,112	37,937,003	34,464,561
768x768	61,326,571	76,464,852	69,720,029
1024x1024	103,102,298	127,850,154	116,774,156
Primary, Shadow K-d			
256x256	50,850,809	60,904,499	56,552,214
512x512	162,613,907	190,557,369	178,034,182
768x768	328,693,749	380,471,543	356,665,276
1024x1024	547,213,764	628,242,585	590,481,703
Primary, Shadow BVH			
256x256	42,053,396	49,152,448	45,867,812
512x512	134,579,333	151,825,923	143,868,974
768x768	275,657,047	308,630,425	293,040,132
1024x1024	465,684,433	517,279,362	493,029,710
Primary, Shadow, Reflect K-d			
256x256	51,090,413	60,552,907	56,394,823
512x512	162,830,975	189,252,084	177,211,481
768x768	328,615,895	377,573,370	354,734,615
1024x1024	546,485,778	623,290,065	587,025,135
Primary, Shadow, Reflect BVH			
256x256	42,552,714	48,843,412	45,749,832
512x512	134,693,538	150,047,767	142,671,755
768x768	274,664,809	304,336,491	289,762,925
1024x1024	462,976,395	509,308,775	486,769,533

Table 5.45: Instructions Executed MGF Office

Theatre	**Min**	**Max**	**Avg**
Primary K-d			
256x256	4,745,966	40,903,308	21,052,700
512x512	17,569,671	130,698,856	66,875,047
768x768	38,308,847	264,168,078	134,541,174
1024x1024	67,391,584	441,949,397	223,512,457
Primary BVH			
256x256	10,035,066	39,568,036	20,847,203
512x512	38,698,152	150,353,436	71,791,766
768x768	85,947,617	331,928,713	152,314,081
1024x1024	151,949,123	585,002,923	262,580,169
Primary, Shadow K-d			
256x256	23,321,616	81,713,347	52,114,573
512x512	85,178,041	257,548,321	167,409,981
768x768	184,774,540	520,164,654	338,780,981
1024x1024	322,168,452	867,820,226	564,610,044
Primary, Shadow BVH			
256x256	29,754,531	86,394,063	56,594,810
512x512	112,023,782	307,901,896	187,888,177
768x768	246,659,993	663,259,074	391,361,900
1024x1024	434,303,177	1,153,353,380	666,869,977
Primary, Shadow, Reflect K-d			
256x256	30,299,378	157,597,159	77,091,884
512x512	106,111,669	497,247,546	245,310,020
768x768	225,448,617	988,644,301	490,600,582
1024x1024	387,058,854	1,618,106,458	808,463,165
Primary, Shadow, Reflect BVH			
256x256	38,835,425	224,463,045	117,063,714
512x512	144,093,745	695,823,205	362,734,548
768x768	314,979,004	1,372,233,168	713,573,324
1024x1024	535,098,839	2,238,891,545	1,162,336,266

Table 5.46: Instructions Executed MGF Theatre

Chapter 6

Conclusions & Future Work

6.1 Conclusion

The goal of this dissertation was to look at the characteristics that K-d trees and BVHs exhibit when used for real-time ray tracing, in particular looking at the traversal of these data structures. To compare these structures I used the codebase from the paper "Ray Tracing on a GPU with CUDA Comparative Study of Three Algorithms" which had a CUDA based implementation of the acceleration structures I required.

From the results of this dissertation we can see that in most cases K-d tree performs better at lower resolutions than the BVH for just the traversal of the tree. The K-d tree also performs better in the dynamic scene in all cases but we must also consider the cost of building or refitting the acceleration structure in which case it is possible that the BVH will end up being faster overall. Overall based on the results of full render experiment in the previous chapter the BVH performs better than the K-d tree.

We can also see in the branch divergence experiment that the K-d tree has around a ten percent divergent branch rate which means there is a potential performance gain to be had whereas the branch divergence percentage with the BVH implementation is very low.

The instruction statistics experiment shows some correlation to the traversal exper-

iment, in most cases the number of instructions executed was higher with the BVH with the exception of with the Stanford Bunny for a number of resolutions.

Finally we come to the dynamic scene experiment, the goal of this experiment was to look at the performance of the acceleration data structure when it is rebuilt every frame. In this experiment the K-d tree outperformed the BVH in all cases. Even though the K-d traversal is faster in this experiment we should also look at the cost of refitting or rebuilding the acceleration data structure with a dynamic scene in this case the BVH could possibly be faster.

6.2 Future Work

Looking towards future work there are a number of experiments that could be carried out or improved upon.

One area to look at would be to render at an even higher resolution such as 1080P (1920x1080) as this is the standard at the moment for real-time gaming applications (With just under 32 percent of gamers running at that resolution based on the steam hardware survey [34]). An even more challenging task for real-time ray-tracing would be to render at the upcoming 4K (3840x2160) resolution, in these particular cases the BVH traversal looks like it would be faster then the K-d tree traversal based on the results shown in this dissertation.

The NVidia tools nvprof and NSight provide a large number of performance metrics which I have only touched upon the full list of available metrics. Some of the metrics I think would be interesting to look at would be SM activity which shows how much each SM is being utilized, a more indepth look at memory statistics such as memory bandwidth between global memory and the caches and finally the achieved floating point operations per second (FLOPS). These are only a small number of the available metrics, the full list of metrics can be seen in the appendix table A.1 or by running 'nvprof –query-events' from the command line.

In the branch divergence experiment we found that there was around a 13% branch

divergence rate for the K-d tree in places. It would be interesting to look at this in detail to find out where exactly this occurs and if it can be optimized to gain a performance improvement.

With new GPU architectures being released every couple of years it would also be interesting to see the performance change. Some research can currently be done here as the GTX 780 or the GTX titan are now available which are based on the newer Kepler architecture whereas the research in this dissertation was based on a Fermi based GPU.

Appendix A

List of nvprof Metrics

sm_cta_launched	Number of thread blocks launched on a multiprocessor.
l1_local_load_hit	Number of cache lines that hit in L1 cache for local memory load accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
l1_local_load_miss	Number of cache lines that miss in L1 cache for local memory load accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
l1_local_store_hit	Number of cache lines that hit in L1 cache for local memory store accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
l1_local_store_miss	Number of cache lines that miss in L1 cache for local memory store accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.

l1_global_load_hit	Number of cache lines that hit in L1 cache for global memory load accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
l1_global_load_miss	Number of cache lines that miss in L1 cache for global memory load accesses. In case of perfect coalescing this increments by 1, 2, and 4 for 32, 64 and 128 bit accesses by a warp respectively.
uncached_global_load_transaction	Number of uncached global load transactions. Increments by 1 per transaction. Transaction can be 32/64/96/128B.
global_store_transaction	Number of global store transactions. Increments by 1 per transaction. Transaction can be 32/64/96/128B.
l1_shared_bank_conflict	Number of shared bank conflicts caused due to addresses for two or more shared memory requests fall in the same memory bank. Increments by N-1 and 2*(N-1) for a N-way conflict for 32 bit and 64bit shared memory accesses respectively.
tex0_cache_sector_queries	Number of texture cache requests. This increments by 1 for each 32-byte access.
tex0_cache_sector_misses	Number of texture cache misses. This increments by 1 for each 32-byte access.
elapsed_cycles_sm	Elapsed clocks
fb_subp0_read_sectors	Number of DRAM read requests to sub partition 0, increments by 1 for 32 byte access.
fb_subp1_read_sectors	Number of DRAM read requests to sub partition 1, increments by 1 for 32 byte access.
fb_subp0_write_sectors	Number of DRAM write requests to sub partition 0, increments by 1 for 32 byte access.

fb_subp1_write_sectors	Number of DRAM write requests to sub partition 1, increments by 1 for 32 byte access.
l2_subp0_write_sector_misses	Number of write misses in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_write_sector_misses	Number of write misses in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_sector_misses	Number of read misses in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_sector_misses	Number of read misses in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_write_sector_queries	Number of write requests from L1 to slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_write_sector_queries	Number of write requests from L1 to slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_sector_queries	Number of read requests from L1 to slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_sector_queries	Number of read requests from L1 to slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_tex_sector_queries	Number of read requests from Texture cache to slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_tex_sector_queries	Number of read requests from Texture cache to slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_hit_sectors	Number of read requests from L1 that hit in slice 0 of L2 cache. This increments by 1 for each 32-byte access.

l2_subp1_read_hit_sectors	Number of read requests from L1 that hit in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_tex_hit_sectors	Number of read requests from Texture cache that hit in slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_tex_hit_sectors	Number of read requests from Texture cache that hit in slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_read_sysmem_sector_queries	Number of system memory read requests to slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_read_sysmem_sector_queries	Number of system memory read requests to slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_write_sysmem_sector_queries	Number of system memory write requests to slice 0 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp1_write_sysmem_sector_queries	Number of system memory write requests to slice 1 of L2 cache. This increments by 1 for each 32-byte access.
l2_subp0_total_read_sector_queries	Total read requests to slice 0 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access.
l2_subp1_total_read_sector_queries	Total read requests to slice 1 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access.

l2_subp0_total_write_sector_queries	Total write requests to slice 0 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access.
l2_subp1_total_write_sector_queries	Total write requests to slice 1 of L2 cache. This includes requests from L1, Texture cache, system memory. This increments by 1 for each 32-byte access.
gld_inst_8bit	Total number of 8-bit global load instructions that are executed by all the threads across all thread blocks.
gld_inst_16bit	Total number of 16-bit global load instructions that are executed by all the threads across all thread blocks.
gld_inst_32bit	Total number of 32-bit global load instructions that are executed by all the threads across all thread blocks.
gld_inst_64bit	Total number of 64-bit global load instructions that are executed by all the threads across all thread blocks.
gld_inst_128bit	Total number of 128-bit global load instructions that are executed by all the threads across all thread blocks.
gst_inst_8bit	Total number of 8-bit global store instructions that are executed by all the threads across all thread blocks.
gst_inst_16bit	Total number of 16-bit global store instructions that are executed by all the threads across all thread blocks.
gst_inst_32bit	Total number of 32-bit global store instructions that are executed by all the threads across all thread blocks.

gst_inst_64bit	Total number of 64-bit global store instructions that are executed by all the threads across all thread blocks.
gst_inst_128bit	Total number of 128-bit global store instructions that are executed by all the threads across all thread blocks.
local_load	Number of executed load instructions where state space is specified as local, increments per warp on a multiprocessor.
local_store	Number of executed store instructions where state space is specified as local, increments per warp on a multiprocessor.
gld_request	Number of executed load instructions where the state space is not specified and hence generic addressing is used, increments per warp on a multiprocessor. It can include the load operations from global,local and share state space.
gst_request	Number of executed store instructions where the state space is not specified and hence generic addressing is used, increments per warp on a multiprocessor. It can include the store operations to global,local and share state space.
shared_load	Number of executed load instructions where state space is specified as shared, increments per warp on a multiprocessor.
shared_store	Number of executed store instructions where state space is specified as shared, increments per warp on a multiprocessor.
branch	Number of branch instructions executed per warp on a multiprocessor.

divergent_branch	Number of divergent branches within a warp. This counter will be incremented by one if at least one thread in a warp diverges (that is, follows a different execution path) via a conditional branch.
warps_launched	Number of warps launched on a multiprocessor.
threads_launched	Number of threads launched on a multiprocessor.
active_warps	Accumulated number of active warps per cycle. For every cycle it increments by the number of active warps in the cycle which can be in the range 0 to 48.
active_cycles	Number of cycles a multiprocessor has at least one active warp.
prof_trigger_00	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
prof_trigger_01	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
prof_trigger_02	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
prof_trigger_03	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
prof_trigger_04	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
prof_trigger_05	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.

prof_trigger_06	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
prof_trigger_07	User profiled generic trigger that can be inserted in any place of the code to collect the related information. Increments per warp.
inst_issued	Number of instructions issued including replays.
inst_executed	Number of instructions executed, do not include replays.
thread_inst_executed_0	Number of instructions executed by all threads, does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 0.
thread_inst_executed_1	Number of instructions executed by all threads, does not include replays. For each instruction it increments by the number of threads in the warp that execute the instruction in pipeline 1.
atom_count	Number of warps executing atomic reduction operations for thread-to-thread communication. Increments by one if at least one thread in a warp executes the instruction
gred_count	Number of warps executing reduction operations on global and shared memory. Increments by one if at least one thread in a warp executes the instruction

Table A.1: nvprof events

Bibliography

[1] Akeley, K., Kirk, D., Seiler, L., Slusallek, P., and Grantham, B. When will ray-tracing replace rasterization? In *ACM SIGGRAPH 2002 conference abstracts and applications* (New York, NY, USA, 2002), SIGGRAPH '02, ACM, pp. 86–87.

[2] Campana, R. S3478 - debugging cuda kernel code with nvidia nsight visual studio edition. In *GPU Technology Conference* (2013).

[3] Christensen, P. H., Fong, J., Laur, D. M., and Batali, D. Ray Tracing for the Movie 'Cars'. *Symposium on Interactive Ray Tracing 0* (2006), 1–6.

[4] Cook, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[5] Ericson, C. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[6] Farber, R. *CUDA Application Design and Development*, 1st ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[7] Foley, T., and Sugerman, J. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), HWWS '05, ACM, pp. 15–22.

[8] Gnther, J., Popov, S., Seidel, H.-P., and Slusallek, P. Realtime ray tracing on gpu with bvh-based packet traversal, 2007.

[9] Goodwin, D. Optimizing application performance with cuda profiling tools. In *GPU Technology Conference* (2012).

[10] Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007* (Sept. 2007), pp. 113–118.

[11] Han, T. D., and Abdelrahman, T. S. Reducing branch divergence in gpu programs. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units* (New York, NY, USA, 2011), GPGPU-4, ACM, pp. 3:1–3:8.

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

[13] Caustic series2 raytracing acceleration boards, 2013. https://caustic.com/series2/index.html.

[14] Quake wars* gets ray traced, 2012. http://software.intel.com/en-us/articles/quake-wars-gets-ray-traced.

[15] Wolfenstein ray traced, 2012. http://wolfrt.de/.

[16] Ize, T., Wald, I., and Parker, S. Ray tracing with the bsp tree. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on* (2008), pp. 159–166.

[17] Laboratory, L. B. N. Mgf parser and examples, 1996. http://radsite.lbl.gov/mgf/.

[18] Lauterbach, C., Garl, M., Sengupta, S., Luebke, D., and Manocha, D. Fast bvh construction on gpus. In *In Proc. Eurographics 09* (2009).

[19] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. Fast bvh construction on gpus. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[20] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News 38*, 3 (June 2010), 451–460.

[21] Lext, J., Assarsson, U., and Moller, T. A benchmark for animated ray tracing. *Computer Graphics and Applications, IEEE 21*, 2 (2001), 22–31.

[22] Moore, D. Fun with kd-trees, 2005. http://onepartcode.com/main/projects.

[23] Nguyen, H. *Gpu gems 3*, first ed. Addison-Wesley Professional, 2007.

[24] Nvidia. *Fermi Compute Architecture Whitepaper.*

[25] Nvidia. *TUNING CUDA APPLICATIONS FOR KEPLER.*

[26] Thinking parallel, part i: Collision detection on the gpu, 2012. https://developer.nvidia.com/content/thinking-parallel-part-i-collision-detection-gpu.

[27] Thinking parallel, part ii: Tree traversal on the gpu, 2012. https://developer.nvidia.com/content/thinking-parallel-part-ii-tree-traversal-gpu.

[28] NVidia. Cuda c programming guide, 2013. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[29] Geforce gtx titan, 2013. http://nvidianews.nvidia.com/Releases/NVIDIA-Introduces-GeForce-GTX-TITAN-DNA-of-the-World-s-Fastest-Supercomputer-Powered-by-World-s-Fa-925.aspx.

[30] Nvidia nsight visual studio edition, 2013. https://developer.nvidia.com/nvidia-nsight-visual-studio-edition.

[31] Optix, 2013. http://developer.nvidia.com/optix.

[32] Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum 26*, 3 (Sept. 2007), 415–424. (Proceedings of Eurographics).

[33] Stanford. The stanford 3d scanning repository, 2011. http://graphics.stanford.edu/data/3Dscanrep/.

[34] Steam. Steam hardware and software survey: July 2013, 2013. http://store.steampowered.com/hwsurvey.

[35] Stich, M., Friedrich, H., and Dietrich, A. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 7–13.

[36] Torres, Y., Gonzalez-Escribano, A., and Llanos, D. Understanding the impact of cuda tuning techniques for fermi. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on* (2011), pp. 631–639.

[37] Zhou, K., Hou, Q., Wang, R., and Guo, B. Real-time kd-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers* (New York, NY, USA, 2008), SIGGRAPH Asia '08, ACM, pp. 126:1–126:11.

[38] Zlatuska, M., and Havran, V. Ray Tracing on a GPU with CUDA – Comparative Study of Three Algorithms. In *Proceedings of WSCG'2010, communication papers* (Feb 2010), pp. 69–76.