TASK PARALLELISM FOR RAY TRACING ON A GPU CLUSTER

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÇAĞLAR ÜNLÜ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

FEBRUARY 2008

Approval of the thesis:

TASK PARALLELISM FOR RAY TRACING ON A GPU CLUSTER

submitted by **ÇAĞLAR ÜNLÜ** in partial fullfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, **Graduate School of Natural and Applied Sciences**

Prof. Dr. Volkan Atalay
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Veysi İşler
Supervisor, **Computer Engineering Dept., METU**

Dr. Cevat Şener
Co-supervisor, **Computer Engineering Dept., METU**

Examining Committee Members:

Assist. Prof. Dr. Tolga Can
Computer Engineering Dept., METU

Assoc. Prof. Dr. Veysi İşler
Computer Engineering Dept., METU

Assist. Prof. Dr. Tolga Çapın
Computer Engineering Dept., Bilkent University

Dr. Onur Tolga Şehitoğlu
Computer Engineering Dept., METU

Dr. Tuğba Taşkaya Temizel
Information Systems Dept., METU

Date: 04.02.2008

ABSTRACT

TASK PARALLELISM FOR RAY TRACING ON A GPU CLUSTER

Ünlü, Çağlar
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. Veysi İşler
Co-Supervisor: Dr. Cevat Şener

February 2008, 51 pages

Ray tracing is a computationally complex global illumination algorithm that is used for producing realistic images. In addition to parallel implementations on commodity PC clusters, recently, Graphics Processing Units (GPU) have also been used to accelerate ray tracing. In this thesis, ray tracing is accelerated on a GPU cluster where the viewing plane is divided into unit tiles. Slave processes work on these tiles in a task parallel manner which are dynamically assigned to them. To decrease the number of ray-triangle intersection tests, Bounding Volume Hierarchies (BVH) are used. It is shown that almost linear speedup can be achieved. On the other hand, it is observed that API and network overheads are obstacles for scalability.

Keywords: Ray Tracing, GPU, Cluster, Task Parallelism, Bounding Volume Hierarchy

ÖZ

ÇİZGE İŞLEM BİRİMİ KÜMESİ ÜZERİNDE IŞIN İZLEME İÇİN GÖREV KOŞUTLUĞU

Ünlü, Çağlar
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Veysi İşler
Ortak Tez Yöneticisi: Dr. Cevat Şener

Şubat 2008, 51 sayfa

Işın izleme, gerçekçi görüntüler üretmek için kullanılan ve hesaplama bakımından pahalı olan bir genel aydınlatma algoritmasıdır. Kişisel bilgisayarlar üzerine kurulan kümelere ek olarak, son zamanlarda çizge işlem birimlerinde de ışın izleme hızlandırılmıştır. Bu tezde, ışın izleme bir çizge işlem birimi kümesi üzerinde görüntü düzlemi birim döşelere bölünerek hızlandırılmıştır. Köle işlemler, kendilerine devingen olarak atanan döşeler üzerinde görev koşutluğu ile çalışmaktadır. Işın-üçgen kesişimi testlerinin sayısını azaltmak için saran hacim hiyerarşileri kullanılmıştır. Neredeyse doğrusal hızlanmaların elde edilebileceği gösterilmiştir. Diğer taraftan, uygulama programla arayüzü ve ağ ek yüklerinin ölçeklenebilirliğe engel oldukları gözlemlenmiştir.

Anahtar Kelimeler: Işın İzleme, Çizge İşlem Birimi, Küme, Görev Koşutluğu, Saran Hacim Sıradüzeni

ACKNOWLEDGMENTS

I would like to express my utmost gratitude and respect to my supervisor Assoc. Prof. Dr. Veysi İşler and cosupervisor Dr. Cevat Şener. Their patience, vision, sweet communication and friendly approach is the key reason to vitalize this work.

I am grateful to Alphan Es and Tümer Topcu, who showed me direction when I was stuck and thought I couldn't break free. They always provided me with insightful comments and helpful hands.

I am also indebted to Yasemin Yardımcı Çetin and Informatics Institute for providing me with the hardware necessary for the experiments in this thesis.

I would like to express my special appreciation to my colleagues Alper Kuş and Ebubekir Tipi who made it possible for me to continue my study even in the busiest days of my job.

I would also like to express my heart-felt thanks to Ayşem, my dear love. Without her unconditional love, joy and support, this thesis could not be completed.

Finally, I would like to thank my parents and grandparents for their everlasting love and support.

To Ayşem...

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

CHAPTER 1

INTRODUCTION

Producing realistic looking images is an important goal of computer graphics. It may take significant amount of time when a realistic looking image is produced by global illumination algorithms such as ray tracing. On the other hand, Graphics Processing Units (GPU) accelerate such production process by the support of specialized processors. However, GPUs support the conventional rendering pipeline which results in less realistic looking images. With the recent advances in hardware, GPUs have gained flexibility and latest GPUs are programmable but they still need to obey some restrictions. Also, the computational power of GPUs have increased a lot and are now exceeding CPUs. For example, a single GeForce 8800GTX can operate at 345 GFlops with 96 parallel processors [41] while Intel has announced a performance of 45 GFlops from 4 dual-core Itanium processors [18]. The raw performance difference between the two architectures speaks for itself. It's also worth mentioning the speed with which GPUs and CPUs improve. While CPU speed doubles every 18 months, GPU speed doubles every 6 months [24]. Considering this, researchers have tried to use the computational power of GPUs to overcome the complexity of ray tracing. There are successful ray tracing implementations but they still suffer from the *not so convenient* programming model of GPUs. These obstacles in front of an easier GPU programming will be mentioned in future chapters.

Initial efforts for accelerating ray tracing that go back are parallel ray tracing implementations on personal computers (PC) and specialized clusters. Compared to GPU ray tracing, it is easier to produce parallel ray tracing solutions on clusters but they have some disadvantages. Like every parallel computing implementation, these solutions need to overcome some challenges which include synchronization of processes, network overheads, bottlenecks, etc.

In the following section, an overview of ray tracing is given.

1.1 Ray Tracing

Ray tracing is an algorithm that was found by Turner Whitted in 1980 [40] in an effort to improve the quality of images that were created by ray casting, which was found in 1968 by Arthur Appel [3]. Its purpose is to create realistic looking images with reflections, refractions, shadows and other visual phenomena like the contribution of multiple light sources in the scene. Ray tracing can be used in many fields that require clear visualization and optical accuracy. The drawback however is the computational complexity of the algorithm. A sample ray traced image can be seen in Figure 1.1.

In the basic algorithm, each ray is tested for intersection against all objects. After an

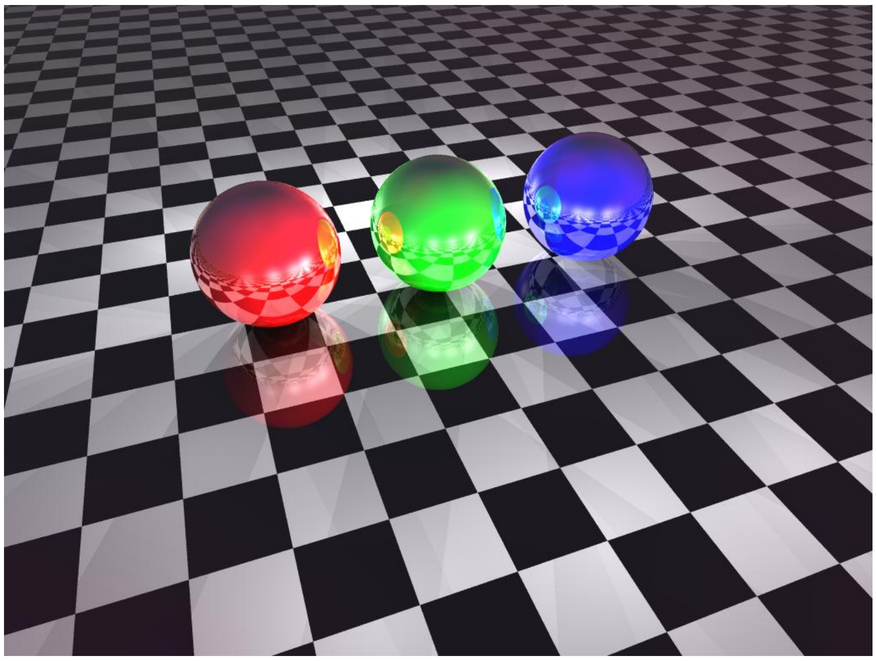


Figure 1.1: Sample ray traced image created by POV-Ray [33]

intersection is found, one ray is generated from that point with respect to the normal if a reflection is present. Also for each light source in the scene, a shadow ray is generated. For transparent objects, yet another ray is generated to account for refraction. Every ray is tested for intersection with objects in the scene, causing the algorithm to require extensive computational resources. Extensions to the algorithm that try to make the resulting image more realistic are computationally even more complex. There have been numerous works to accelerate ray tracing. However, even with the lowered cost, the algorithm is still not

Algorithm 1 Whitted-Style Ray Tracing

```
function render()
    for each pixel do
        generate a ray from the eye point through this pixel
        pixel color ← trace(ray)
    end for


function trace(ray)
    find nearest intersection with ray
    compute intersection point and normal
    color ← shade(point, normal)
    return color


function shade(point, normal)
    color ← 0
    for each light source do
        trace shadow ray to light source
        if shadow ray intersects light source then
            color ← color + direct illumination
        end if
    end for
    if surface is specular then
        generate a new reflection or refraction ray
        color ← color + trace(new ray)
    end if
    return color
```

applicable to real-time dynamic scenes that are frequently seen in computer games and animations. Making ray tracing a viable option for everyday use is a major challenge.

1.1.1 Acceleration Structures

Whitted-style ray tracing can be seen as a pseudo-code in Algorithm 1 [19]. Although the algorithm isn't complete, it demonstrates the fundemental ideas behind ray tracing. The algorithm looks simple, but the number of intersections tests it requires gets out of control very quickly because each ray has to be tested with all triangles in the scene. Some methods have been proposed that decrease the number of intersection tests significantly. These methods cause ray tracing to remain relatively cheap when scenes get larger. To be more specific, when one of these methods are used, ray tracing has logarithmic complexity with respect to the number of triangles in the scene [16]. It's obvious that using acceleration techniques is mandatory when the exponential complexity of the original algorithm is considered.

Below are some of the widely used acceleration structures.

Bounding Volume Hierarchy

A Bounding Volume Hierarchy(BVH) is simply a tree that consists of bounding boxes that get smaller as one goes deeper in the tree. The root of the tree is the bounding box of the whole scene and leaves of the tree are the triangles themselves. To find the intersection of a ray with the triangles in the scene, one starts from the root node, and traverses the tree. If the ray doesn't intersect a bounding box, it is guaranteed that the ray doesn't intersect the children of that bounding box either, at which point, the traversal can continue from the next node.

One important choice when creating a BVH is the characteristic of the bounding boxes. Given the bounding box B of a fixed set of objects and a ray r, the probability of r intersecting B decreases as B gets smaller. Although, there are methods to find complicated bounding boxes to closely fit objects [14], the most widely used type of bounding box is the Axis Aligned Bounding Box (AABB). The reason for this is the convenience of finding ray-AABB intersections.

There are different ways to create a BVH for a given scene. Two popular methods are as follows. Kay & Kajiya [20] have proposed a top-down approach where the triangles are sorted along the axes and an axis is chosen to make the split. The triangles in the current bounding box are split in two and a recursive approach is applied to both sides of the split. The resulting tree is binary by definition. The other method proposed by Goldsmith & Salmon [15] is a bottom-up approach where the triangles are added to the tree one by one with the first triangle in the list becoming the root. Remaining triangles are added to the tree using a cost function. They are positioned either as leaves or by introducing new intermediate nodes that shift an existing subtree one level deeper, having the new triangle as a sibling to this subtree. The resulting tree for this approach does not have to be binary.

Uniform Grid

Uniform Grid is a method that partitions the space. After finding the AABB of the scene, the bounding box is divided into a three dimensional grid and triangles are assigned to voxels that contain them. The traversal is pretty straight forward as a 3D DDA algorithm can be used to find the next voxel and the ray is tested against the triangles in that voxel. One advantage of uniform grid is that as soon as an intersection is found, the traversal can stop since ray-triangle intersection tests start from the voxels that are closest to ray origins [38].

KD-Tree

KD-Tree is yet another spatial division method for fast traversal of space. It is a binary tree with arbitrarily sized children. The construction of kd-tree resembles the construction of BVH that is proposed by Kay & Kajiya. It's a top-down approach, in which, as construction goes deeper in the tree, splitting axis is chosen by cycling the axes. After deciding the splitting axis, space is divided into two instead of calculating the bounding boxes of the triangles. To traverse a kd-tree, one starts from the root node. If the ray intersects both of the children, the nearest one is chosen for intersection test and the other one is pushed to the stack. If the ray intersects only one child, the process continues with that node without pushing anything to the stack. When the current node fails the intersection test, stack is popped and the process continues with the popped node. The process ends when a leaf node has a triangle that intersects with the ray or when the stack is empty and there are no more nodes to process [12].

Octree

Octree is also a spatial division method in the form of a tree. It is a special type of kd-tree. The construction of the octree begins by finding the bounding box of the scene and recursively dividing each node into eight identical volumes in a top-down approach. The decision of identifying a node as leaf depends on the number of triangles in the node or the depth of the node. To traverse the octree, different methods have been proposed that more or less resemble uniform grid or kd-tree traversals [38].

1.2 General Purpose computation on GPUs (GPGPU)

Until recently, GPUs were only good at rendering simple primitives and it wasn't possible to customize the processors inside. With the latest advances in hardware, GPUs have become faster and more flexible. Current GPUs are highly parallel processors that are designed for graphics applications. They are not as sophisticated as CPUs but they surely are more powerful, assuming that they are used for what they were created for, which is making intense floating point arithmetic. With limited branching capabilities and sub-optimal random read and write operations, there is one additional reason that causes programmers to not be able to use them to their full capacity. Because GPUs were designed for graphics applications that use the forward rendering pipeline, the most popular way to customize them is through the use of a graphics API. However the interference of such an API brings some restrictions and overheads, which decreases the utilization of GPU power.

1.2.1 GPU Programming Model

In rasterization based rendering, the scene, which consists of objects in 3D, can be represented in various ways. One of the most popular and convenient ways is keeping the scene as a triangle mesh. In a mesh, the main data element is the point. A mesh consists of a number of points in space and the connectivity information for edges that connect the points, possibly with some additional information such as normals, materials, textures etc. All this information is used in the shading process which determines the final color of each pixel in the resulting image. These meshes can be fed to the standard graphics pipeline (Figure 1.2) to get the rendered 2D image of the scene. The mentioned pipeline is a chain of computations that initially take the mesh as input. Every step of the pipeline processes the input in a predefined way to finally produce the resulting image. With programmable

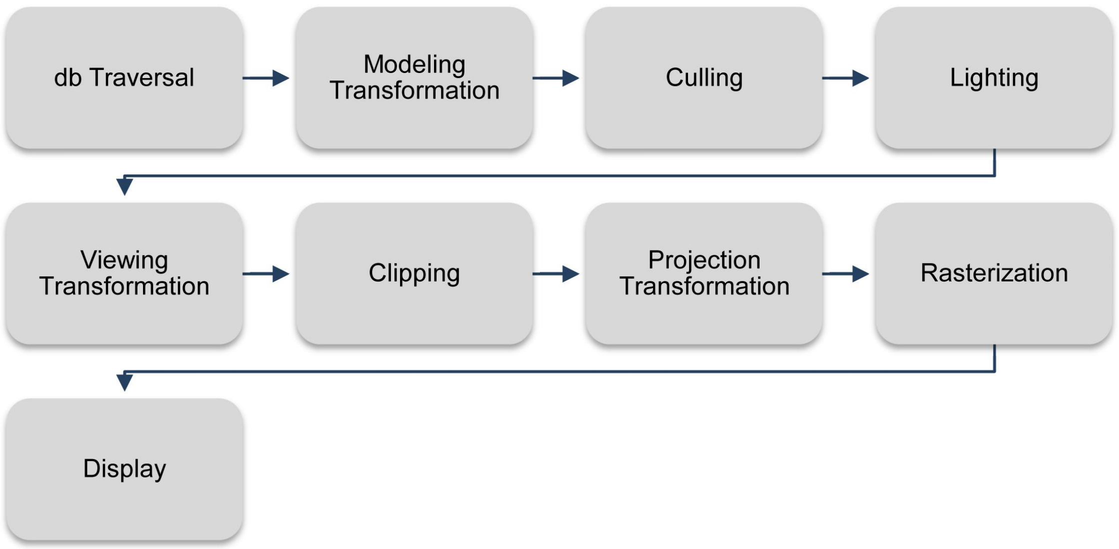


Figure 1.2: Rasterization based rendering pipeline (Courtesy of Alphan Es [7])

graphics hardware flooding the market, the standard pipeline has evolved into what can be seen in Figure 1.3. Some steps have disappeared, while some new steps have emerged with some others dissolving into other steps. The green steps in the pipeline denote the programmable parts. Figure 1.4 shows how the GPU pipeline is used practically. Custom programs are written for Vertex, Geometry and Fragment Processers that suit the user's computational needs. The input is fed to the pipeline as well as texture. Vertices are processed in the Vertex Processor followed by the Geometry Processor. Finally the Fragment Processor prepares the final image by adding shading and the result is handled by the Output Merger which handles blending, depth test, stencil test etc.

Up to this point, the section has shown how the GPU pipeline works for normal graphics applications with some customization by using different processors throughout the pipeline such as Vertex, Geometry and Fragment Processors. From this point on, the document will point out how the GPU can be used for general purpose computing. Even though the subject of this thesis is a graphics algorithm, the actual computations that are carried out on the GPU actually resemble general purpose computing rather than graphics rendering. As such, some introduction to stream processing is needed before passing on to details about GPU ray tracing.

Figure 1.5a shows how stream processing works. A kernel is a program that takes a set of streams as input and outputs another set of streams. Figure 1.5b shows where kernels are positioned in the GPU pipeline. The green boxes are kernels. In GPU programming, input

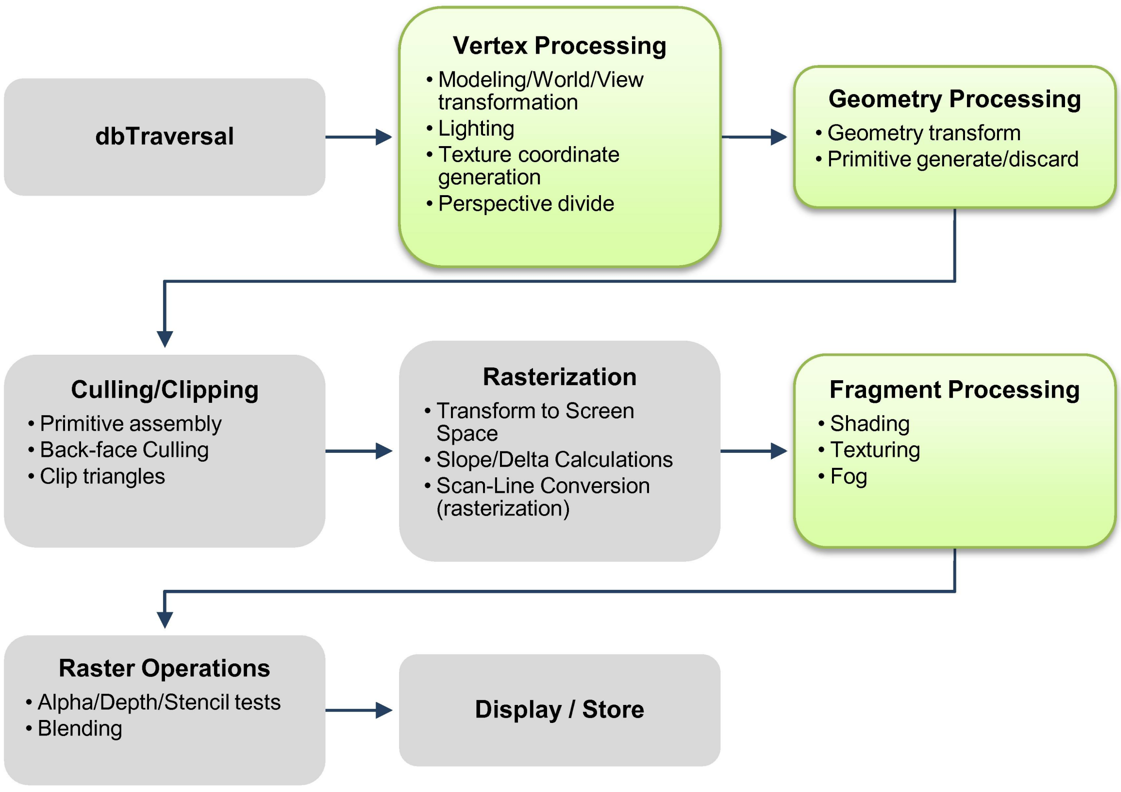


Figure 1.3: GPU Pipeline (Courtesy of Alphan Es [7])

streams can be vertex data (position, normals), textures or fragments and output streams can be geometry or fragments [7]. In a single pass, the GPU executes vertex, geometry and fragment kernels once. The output of one kernel can be an input for one of the other kernels or even the kernel that output it in a consequent pass. To execute a specific kernel twice, however, the whole pipeline must be traversed, meaning, it's not possible to run a fragment program twice without running a vertex program twice too. Also worth notice is the fact that the whole pipeline is traversed for the whole render target. This means, even if 99% of the pixels don't need processing, they are still passed as inputs to various kernels. However, there is a way around this which is called Early Fragment Culling. Early Fragment Culling speeds up the process when some pixels may get calculated in less number of passes. It works by distinguishing between pixels that need processing and those that do not and rejecting the finished pixels even before the kernel is called for execution. To do this, Depth or Stencil information should be updated between passes to indicate which pixels still need computation.

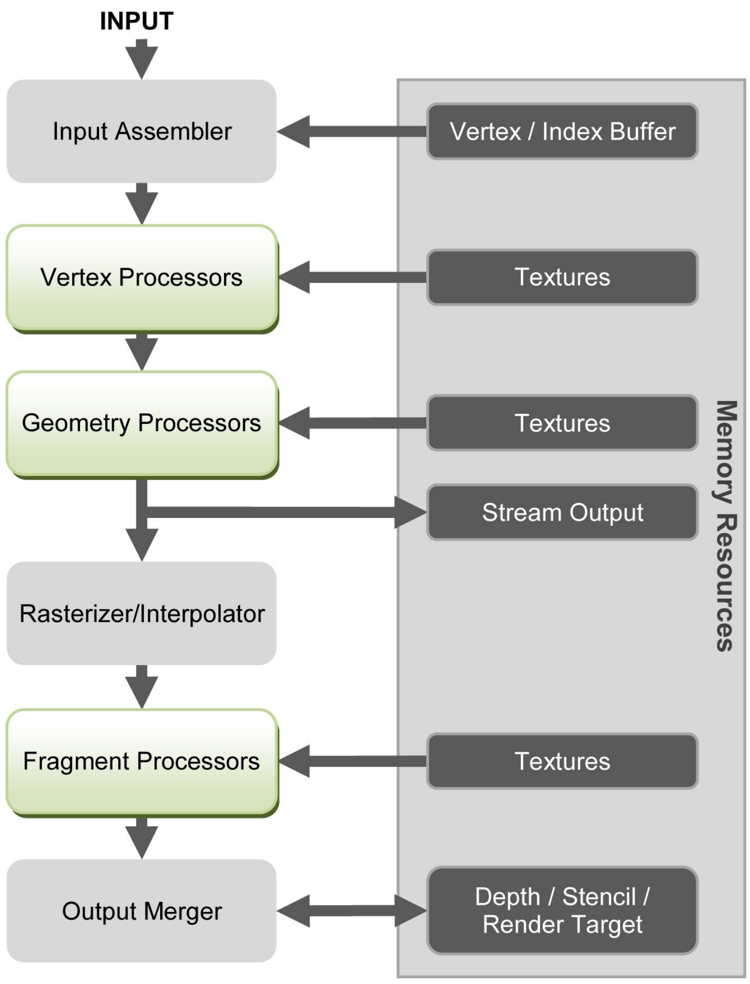
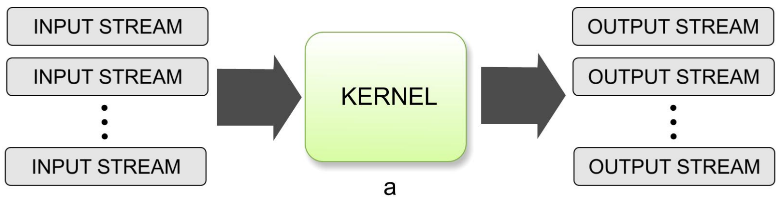


Figure 1.4: GPU Processing (Courtesy of Alphan Es [7])

1.2.2 Programming GPUs

GPUs can be programmed using graphics APIs such as OpenGL [32] or DirectX [25]. These APIs offer assembly-like languages for programming the GPU. However, another method, highly recommended, is using high level languages. By using high level languages, the user doesn't have to worry about low level optimizations dependent on hardware and instead, can focus on the task at hand. Since porting general purpose problems to GPU is no simple task, using high level languages makes it at least a little simpler to design solutions for GPUs.

High level languages include but are not limited to NVidia's Cg (C for Graphics) [29] and CUDA [28]. While Cg supports OpenGL and Direct3D, these two APIs also have their own high level languages, namely GLSL (OpenGL Shading Language) [30] and HLSL (High Level Shading Language) [26]. CUDA on the other hand, is a toolkit that lets devel-



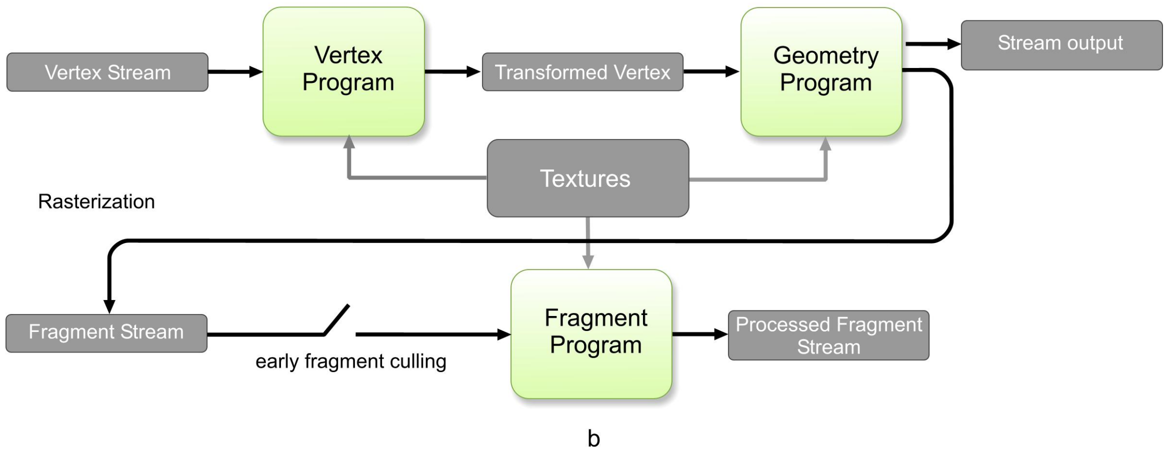


Figure 1.5: (a) Stream Processing (b) Stream Processing on GPU (Courtesy of Alphan Es [7])

opers use C to program GPUs and relieves them of cumbersome graphic APIs which are widely used.

Brook for GPUs is developed by Stanford Computer Graphics Laboratory and is a compiler and runtime implementation for the Stream Processing Language Brook [21]. It is aimed for using on GPUs. Another such language is Sh which is aimed for both GPUs and CPUs [17].

There is also CTM (Close-to-the-metal) which is being developed for AMD stream processors. It is claimed that the performance is eight times better than a traditional API for GPU programming purposes [2].

1.3 Parallel Computing

Most software developed until now works on a single CPU in a sequential manner (serial computing). There are instructions one after another, which are executed one at a time. This way, a program cannot get much faster. Assuming a difficult problem that can be divided into subproblems, solving them concurrently and merging the solutions at the end could yield the solution to the original problem. Ideally, this would decrease the time to solve the problem. However, not all problems can be solved in parallel. To parallelize the solution to a problem, the problem needs to be dividable into smaller problems, and it should take less time to solve each subproblem. One other property that favors parallelism is having little dependency. Dependence is said to exist if changing the order of execution of two parts of the program changes the result. Ray tracing is a good example for a problem that has almost no dependence because the order of execution for different pixels doesn't effect the resulting image. That is, the computations for the final value of pixel A can be done before the computations of pixel B, and vice versa where A and B are any pixel on the screen.

If one is considering parallel computing, there are two laws that should be known, which, more or less serve the same purpose. The purpose of these laws is determining how much parallelism can be achieved for a given problem. The laws are given below [42].
Amdahl's Law: $S = 1/(1 - P)$ where S is the speedup of the program and P is the fraction of the problem that can be parallelized.
Gustafson's Law: $S(P) = P - \alpha \cdot (P - 1)$ where S is the speedup of the program, P is the number of processors and α is the fraction of the problem that can't be parallelized.

The difference between these two laws is that Gustafson's Law doesn't assume that the ratio of the parallelizable part is fixed, but instead, it can change with the number of processors involved. For example, according to Amdahl's Law, if the part of the problem that can be parallelized makes up 80% of the problem, then, parallelizing that problem can yield a speedup of at most 5. The same problem, according to Gustafson's Law, would yield at most 3.4 speedup with 4 computers and 6.6 speed up with 8 computers. Of course, these values represent the ideal speedup of parallelizing a problem. In practice, implementations that are even slightly close to the ideal are considered successful. The real speedup of a parallel solution is measured by $Speedup(n) = T(1)/T(n)$ where $T(1)$ is the time it takes to run the serial implementation and $T(n)$ is the time it takes to run the parallel implementation on n computers. Ideally, this ratio should be as close as possible to n which means that a *scalable* application should get almost n times faster if it's run on n computers. However, adding more computers to run the program faster isn't always the best solution. With network latencies, synchronization overheads, memory management problems and possibly a large fraction of the problem that can't be parallelized, a program may not be performing as well as expected beyond a critical number of computers. At this point, the optimal number of computers should be determined for the problem at hand.

Load Balancing is another important factor that determines the success of a parallel program. The processes that are involved in the parallel computing should share the work as well as possible. Decreasing the time processes spend in idle also decreases the time with which the computations get done. One of the easiest methods of load balancing is to execute different parts of a loop in different processes. This gives the intended result if each iteration has the same amount of work to be done. However, the time spent for executing a given part of the loop may vary between processes if the workload for each iteration is not the same. Thus, some other way to balance the load on different processes is needed. Dynamically assigning tasks to processes on-the-fly can be one way of doing this but it requires more delicate planning.

Level of granularity is also important and it should be decided when a parallel solution is designed. For almost all communication calls, computation stops for a while for synchronization between communicating processes. Thus, most of the time, passing big messages less frequently can be more efficient. Of course, this is not the best way to handle communication all the time because load balancing is harder when this approach is used [23].

One other choice that has to be made is about the type of memory that will be used. One can choose to use shared memory or distributed memory or even the hybrid of the two. If shared memory is used, every processor has access to a global memory. While this may reduce the need for explicit communication between processes [23], it requires the programmer to handle memory access by taking critical sections into consideration [8]. Distributed memory systems, on the other hand, don't need to bother with the technicalities of shared memory systems, but they require much more communication which introduces network overheads to the program. Hybrid systems try to make use of the better parts of both methods depending on the problem at hand. One could say that shared memory systems need better memory management while systems that use distributed memory need better communication.

One popular way of classifying parallel computing is Flynn's Taxonomy which was proposed by Michael J. Flynn in 1966 [23]. The taxonomy can be seen in Table 1.1. SISD systems consist of a single CPU that executes instructions one after another. There is no parallelism whatsoever. PCs are a good example for this (excluding ones with more than one processor which have emerged lately). In SIMD systems, there is a single set of instructions that run on some part of the data. Examples for this are GPUs and array processors. MISD systems have different instructions streams that work on the same set of data. Some

Table 1.1: Flynn's Taxonomy

SISD Single Instruction Single Data	SIMD Single Instruction Multiple Data
MISD Multiple Instruction Single Data	MIMD Multiple Instruction Multiple Data

examples for this rarely seen model is to try to crack a code on different machines that apply different algorithms to the code or to apply different frequency filters to an input stream. MIMD systems have at least two set of instructions that work concurrently on different data. Current multi-cored CPUs are an example to this as well as grids and most of the supercomputers.

There are two ways of decomposing a problem for parallelization. These are data parallelism and task parallelism. In data parallelism, each process works on some distinct part of the data and performs the same work as every other process. The data that different processes work on doesn't overlap. In task parallelism, different tasks can work on the same data or different data performing different tasks [8].

In this thesis, task parallelism is used for sharing the work load between processes. Each process has access to the whole scene and a screen space division takes place to distribute the work. The details of how this approach is employed will be explained in Chapter 3. However, data parallelism could also be used for parallelizing ray tracing on a cluster. In a data parallel approach, each process has access to only some parts of the scene. This causes processes to test rays only against the triangles that they have access to. When a ray enters a part of the scene that is not accessible by that process, it is transferred to a process that has access to the mentioned parts.

For ray tracing, task parallelism is easier to implement and more successful compared to data parallelism. Efficient implementations can easily reach linear speedups [5]. On the other hand, task parallelism requires that the memory that each process has access to is big enough to hold the whole scene. Thus, task parallelism fails at rendering big scenes. Data parallelism is the better choice for such scenes since, by definition, scene data is divided to processes and the maximum size of the scene that can be ray traced is limited by the number of computers. One drawback of this approach is that data parallel ray tracing implementations require very efficient communication and carefully planned synchronization. They also suffer from load imbalances.

1.4 Motivation Behind the Thesis

The motivation behind this thesis, like many others of its kind, is to accelerate ray tracing. This is done using a small GPU cluster. Since pixels are independent of each other in ray tracing, issuing groups of pixels to each process appears to be the obvious choice. Thus, the cluster is used in a task parallel manner for sharing the workload. To decrease the number of intersection tests, BVH will be used.

1.5 Thesis Outline

Organization of this thesis is as follows, in Chapter 2, a literature survey on parallel ray tracing and GPU ray tracing is provided. Chapter 3 presents the main work of this thesis. After explaining experimental setup, results are given in Chapter 4 with some discussion. Finally, Chapter 5 proposes ways to accelerate ray tracing even further and concludes this work.

CHAPTER 2

RELATED WORK

Some of the studies done that are related to this thesis are mentioned in the following sections.

2.1 GPU Ray Tracing

With the emergence of programmable hardware, there have been many successful attempts at porting the ray tracing algorithm to GPUs. Some of them will be mentioned here to familiarize the reader with the developments in GPU ray tracing.

2.1.1 Carr *et al.*

Carr *et al.* have proposed using the GPU only for making ray-triangle intersection tests instead of porting the whole algorithm to GPU. They call it the ray engine [4]. Coherent rays are tested against triangles using pixel shaders. To accelerate ray tracing, octree is used as well as a 5-D ray tree. They have also utilized the CPU by using the NV_FENCE OpenGL extension so that they could keep the CPU busy by knowing when the GL calls related to GPU computations return. The resulting images have some artifacts due to limited floating point capacity of the GPU they used. The speedup is less than double compared to the CPU implementation. The reason for this is claimed to be the overhead of ray caching which was used to ensure that groups of coherent rays were sent to the GPU.

2.1.2 Purcell

Purcell considered the GPU as a stream processor and converted ray tracing into a stream program. He proposed some abstractions that would make the GPU a stream processor. He identified the restrictions of GPUs and tried to find solutions that would help create a successful stream program which could execute the ray tracing algorithm. During the time Purcell worked on his thesis, GPUs weren't as flexible as they are today. There were limits on the number of instructions in a fragment program and number of texture lookups, the cache that could be used was small and programs couldn't output to multiple targets [34].

Contrary to Figure 1.5a, Purcell's definition of kernel accepts only one stream as input. He defined four kernels for the main tasks in ray tracing. They are eye ray generation, acceleration structure traversal, ray-triangle intersection and shading. The eye ray generator creates one ray for each pixel in the view plane. These rays are given to the traversal kernel as input which finds the next portion of space the ray will go through next. The output of this kernel is given to the ray-triangle intersector to find hits. If a hit is found, the output goes to the shader. If not, the ray is passed back to the traversal kernel for further steps through the scene. The final image is produced by the shading kernel.

2.1.3 Foley & Sugerman

After numerous works that use uniform grid for GPU ray tracing and claims from Havran [16] that imply kd-tree as the best performing acceleration structure for CPU ray tracing, Foley & Sugerman have proposed a method [12] that uses kd-trees without requiring a stack. Ernst *et al.* [9] had managed to implement a stack structure on GPU, however that work had big time and memory requirements for maintaining the stack. Foley & Sugerman propose two algorithms in their work that negate the need for a stack on GPU. The kd-restart algorithm works by restarting the process from the root of the tree when a leaf node doesn't yield any hit and goes down in search of an intersection. The normal search continues when the ray is inside the node that would be at top of the stack if a stack was used. The kd-backtrack algorithm is an improved version of the kd-restart algorithm with somewhat higher space requirements. The nodes in the tree keep a pointer to their parent, and instead of restarting from the root, the algorithm backtracks to the appropriate node. The result is the same with kd-restart with slightly more space requirement and faster initial ray casting. Both algorithms perform similarly well in scenes that resemble the *teapot in a stadium* but are well behind uniform grid in a homogenous scene like the *Stanford Bunny*. Although the results aren't spectecular, the work is important because of the methods it introduces.

2.1.4 Thrane & Simonsen

Thrane & Simonsen have introduced BVH to GPU ray tracing [38] as well as testing the ever-popular uniform grid and the recent kd-tree mentioned in Subsection 2.1.3. In almost every GPGPU application, a fragment program is just a way to produce some output that will later be an input to another fragment program. Thus, instead of displaying the output of the fragment program on the screen, it's stored in an off-screen buffer. Thrane & Simonsen have used pbuffers [31] for this purpose. Nowadays, frame buffer objects [36] are another way of storing the output of a fragment program. To call different programs one after the other as many times as needed, textures in the pbuffer are used alternatingly. During one pass, texture A is used as input and texture B is used for storing the output. During the next pass, texture B is the input and texture A is used for storing the output. This technique is widely used in GPGPU and is named *ping-ponging* [13]. Normally switching between different pbuffers (which is called as a context switch) introduces some unwanted delay. To avoid it, Thrane & Simonsen have tried to use the same buffer as both input and output. Since all pixel computations can be done independent of each other in ray tracing, fragment programs don't need to read pixels other than the one they are working on. Therefore, changing the value of a pixel after it is read doesn't cause any unexpected behavior. Getting rid of the unnecessary context switch saves significant amount of time.

The resulting kd-tree and uniform grid might have triangles that are inside more than one voxel. It is imperative that the correct intersection is found for all rays, thus, a method called mailboxing [1] was investigated for intersecting a ray against all triangles in its path while also making sure that redundant computation wasn't performed by testing the same ray-triangle couple more than once. However, it was decided not to use it after considering shared insights with Purcell on the difficulty of implementing mailboxing on GPU and Havran's evidence that the result was simply not worth it with the occasional slowdown. The biggest reason for the mentioned difficulty is that the *scatter* operation (writing to random memory) in fragment programs is available only indirectly.

For uniform grid, they have used 2D textures to store the scene data for development convenience although it was possible to use 3D textures too. This requires keeping slices of grid in smaller rectangles inside the textures. Instead of keeping the triangles in the grid, they keep a pointer to a list of triangles. The list is in fact another list of pointers that refer to the actual triangles which are kept in another texture. While the need for an extra memory fetch is introduced by this, the space requirement goes down since each triangle is stored only once instead of storing it more than once for each voxel it is in.

The kd-tree implementation used in their work is based on Foley & Sugerman's kd-restart and kd-backtrack algorithm(Subsection 2.1.3).

The biggest contribution from Thrane & Simonsen is their BVH traversal algorithm. The details of this algorithm will be revealed in Chapter 3.

Thrane & Simonsen conclude their work by stating that their BVH algorithm is very efficient. Space and time requirements of the algorithm are very low. Having tried both the bottom-up and top-down methods for creating a BVH tree that was mentioned in Chapter 1, their results show that both methods are superior to other acceleration structures. This contradicts with Havran's work [16] who stated that BVH performed the worst among the methods tested in his work, but his tests were done using CPUs and BVH might actually be better suited for GPUs.

2.1.5 Es & İşler

One of the more recent works on GPU ray tracing is done by Es and İşler [10]. In their work, they have compared four methods based on regular grids including their method. The methods they have chosen for comparison are Amanatides and Woo's digital difference analyzer (DDA) based ray tracer [1], Cohen and Sheffer's proximity cloud (PC) based ray tracer [6] and Sramek and Kaufman's anisotropic chessboard distance (ACD) based ray tracer [37]. They propose an improved version of ACD which they call as the extended anisotropic chessboard distance (EACD). The difference between ACD and EACD is that, instead of holding a single distance information for all three axes, separate distance information for each axis is stored for all eight octants that define ray direction. Thus, the space requirement is three times that of ACD. However, by using packed formats, this cost can be reduced to an extent. This high cost is still an obstacle for increasing the performance. This is due to the fact that EACD favors smaller voxels and the limits for texture size on the GPU are reached when a critical grid resolution is chosen.

The experiments show that EACD is very successful compared to other grid methods. It also seems to be more successful than non-grid based methods that were studied by other researchers. This is especially true in scenes like the *Stanford Bunny* while the performance gain decreases slightly in scenes that don't have a uniform triangle distribution such as the *teapot in a stadium*. Es and İşler also mention that traversal takes twice as long when the screen resolution is quadrupled. The reason for this is claimed to be the increased coherence.

2.2 Parallel Ray Tracing

The computational complexity of ray tracing has made it very hard to use it on a single computer. Even with acceleration structures that reduce the number of ray-triangle intersection tests, results have urged researchers to try parallel implementations which use the computational power of more than one computer. The nature of the problem is very suitable for parallel computing and there is more than one way to implement parallel ray tracing. For small scenes for which the scene database is small, each node can have the scene data stored. However, this is not possible for big scenes where dividing the scene data between nodes seem to be the only choice.

2.2.1 Wald *et al.*

In their work, Ingo *et al.* mention achieving interactive frame rates [39]. Assuming that the application will be used on a single master computer, they have designed their cluster to work in a client/server fashion. Trying to optimize every aspect, they have written their own network libraries instead of using more general purpose parallel computing libraries such as PVM [35] or MPI [27]. To effectively use every tiny interval of time, communication and rendering is done asynchronously which means that the server starts preparing future frames even as the clients are busy computing the current frame. Also, a client doesn't have to wait for a new tile to render, because all clients have, on average, 4 tiles assigned to them. In other words, they can start a new tile as soon as they are finished with the current tile. Another strategy used by Ingo *et al.* is assigning the previously rendered tiles to respective clients in future frames to exploit cache speed on client processors. Once all tiles are done, a client simply tries to render a tile that was normally assigned to another client. Their results are impressive. They have achieved almost linear speedup on a cluster that has 48 nodes, rendering scenes that have more than a billion triangles with up to 24 frames per second. The bottleneck that they hit was the bandwidth of their server node.

2.3 GPU Cluster

In 2004, Fan *e*t al. used a GPU cluster to do some general purpose computing. They used a cluster which consisted of 32 nodes (each of which was equipped with a GeForce FX5800 Ultra) to compute LBM and simulate the transport of airborne contaminants in the Times Square area of New York City [11]. They have achieved a speedup of 4.6 with respect to the same model implemented on a CPU cluster.

CHAPTER 3

THE GPU CLUSTER

In this chapter, the main work of this thesis will be explained. Section 3.1 will mention the construction of the Bounding Volume Hierarchy (BVH) and Section 3.2 will enlighten the reader about the GPU implementation while Section 3.3 will present the details of the parallelization proposed for this work.

3.1 BVH Construction

In this thesis, the BVH that is used for the GPU implementation is built using a top-down approach that is proposed by Kay & Kajiya [20]. The simple modification that is used by Thrane & Simonsen is applied which is to split the current node using the axis which will yield the smallest total area of the two child bounding boxes. Algorithm 2 shows how this is done. To split the triangles, they need to be sorted. To sort them, Combsort11 is used which is a variation of the Combsort Algorithm [22] that was found by Stephen Lacey and Richard Box in 1991. Combsort is based on the Bubblesort algorithm and runs in $O(nlogn)$ time. Tests show that it's not as fast as Quicksort but the stack requirement of Quicksort was an obstacle for even the *Stanford Bunny* (69451 triangles). Unlike Quicksort, Combsort11 doesn't use the stack.

The tests were done using static scenes so the BVH is constructed offline.

3.2 GPU Implementation

For our GPU implementation, Thrane & Simonsen's BVH model [38] is chosen, because of its performance and simplicity. Figure 3.1 demonstrates how their algorithm works. In the figure, numbered circles are nodes of the final bounding volume hierarchy. The numbers inside the nodes are called *sequence indices*. Leaf nodes are single triangles and

Algorithm 2 Kay/Kajiya BVH Construction

```
function BVHNODE BuildTree(list of triangles)
    if there is only one triangle in the list then
        return leaf holding the triangle
    else
        Calculate the best splitting axis and where to split it
        BVNODE $result$
        $result.leftChild$ ← BuildTree(triangles left of split)
        $result.rightChild$ ← BuildTree(triangles right of split)
        $result.boundingBox$ ← bounding box of all given triangles
        return $result$
    end if
```

internal nodes are bounding boxes. The node numbered 11 doesn't represent an actual bounding box. It is only used for ending the traversal. The dashed lines indicate the *escape indices* for the respective node. The purpose of these indices will be explained shortly.

The ray-scene intersection is a depth-first traversal of the tree. The pseudo-code can be seen in Algorithm 3. Each ray is tested against the whole tree. The traversal begins from the root node. If a ray intersects a bounding box or a triangle, the traversal continues from the next node in the tree. To decide the next node to be processed, the current node index is incremented by one. Before passing on to the next node, the intersection information is stored if the current node was a triangle. However, if the intersection test fails, the traversal continues from the node that is indicated by the *escape index* of the current node. The *escape indices* for a sample BVH can be seen in Figure 3.1. The traversal ends when the current index is equal to the number of nodes in the tree. This is depicted by the node 11 in the figure.

It's worth noting some patterns in this tree. The *escape indices* for the leaves are one more than the *sequence index* for that leaf. Thus, all leaf nodes are aware of their *escape indices* without a need for additional storage. Also, the *escape index* for all right children is equal to that of their parent. Finally, all left children have their sibling's *sequence index* as their *escape index*. This also explains why the root node has an *escape index* of 11. Assume there was a parent to nodes 0 and 11. Then, nodes 0 and 11 would be siblings and 11 would

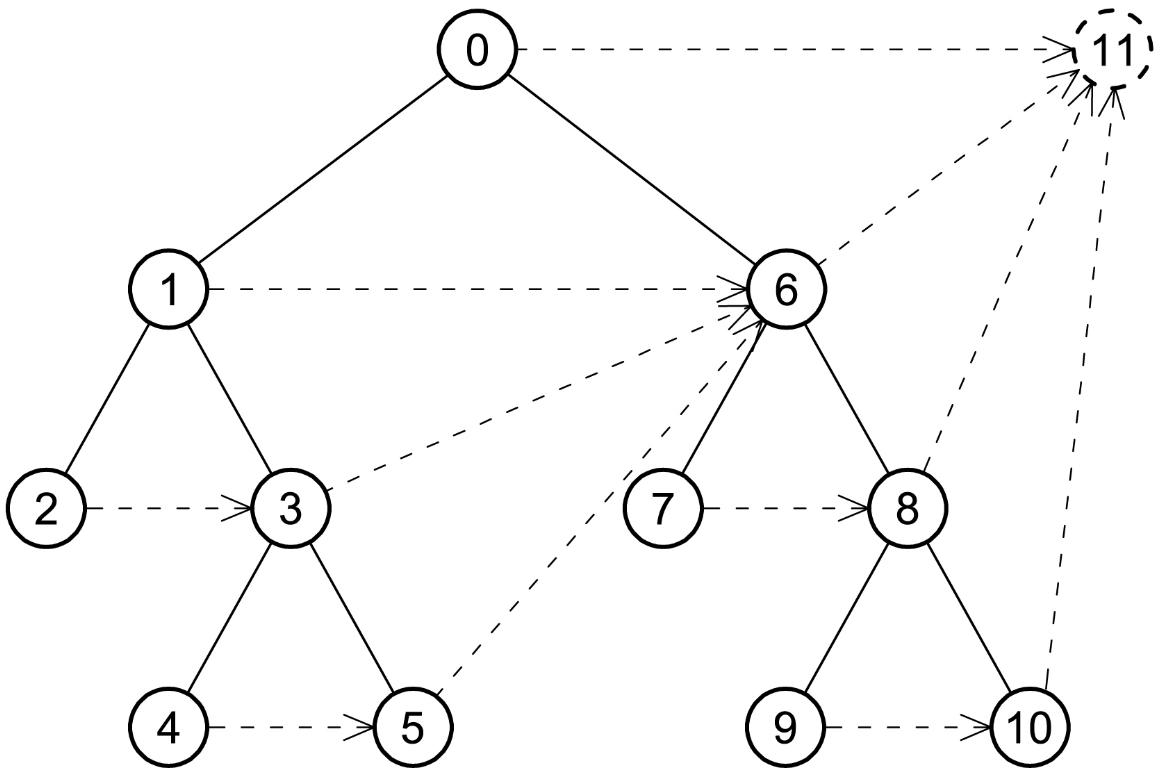


Figure 3.1: Sample Bounding Volume Hierarchy

be the escape index of node 0.

This traversal scheme has some consequences. First of all, it guarantees that the traversal will end, because the index for current node always increases and at some point is more than the maximum valid index. However, the fixed-order traversal also has some disadvantages. When a ray intersects a triangle, there is no way to know if that triangle is the first intersection or not. That is, there may be other triangles that the ray intersects, but to decide which one is the closest to the camera, the whole scene needs to be tested. Even though a large fraction of the tree can be pruned, there are times when most of the tree needs to be traversed. For example, in the worst case, a ray may need to traverse every single node of the tree before finally deciding which triangle was actually hit first. This is different than other methods such as kd-tree or uniform grid, where adjacency information is used for traversing the volumes. Those methods can stop the traversal as soon as an intersection is found because the intersections tests start from voxels that are closest to the camera. Chapter 4 will present evidence about this problem.

To prepare the tree for Algorithm 3, a depth-first pass needs to be performed to set the *escape indices* after the tree is constructed. The pseudo-code for this can can be seen in

Algorithm 3 Thrane & Simonsen GPU traversal algorithm

```
function TraverseBVH
    $S \leftarrow$ The Traversal Sequence
    $r \leftarrow$ The ray
    $currentIndex \leftarrow 0$
    while currentIndex < length of S do
        $currentNode \leftarrow$ element at position $currentIndex$ in $S$
        if $r$ intersects currentNode then
            $currentIndex \leftarrow currentIndex + 1$
            save intersection data if currentNode is a leaf/triangle
        else
            $currentIndex \leftarrow$ escape index of $currentNode$
        end if
    end while
```

Algorithm 4. To start setting the *escape indices*, *BVHRoot.SetEscapeIndex(Number of Nodes)* is called. After this, another depth-first pass is performed to write the contents of the tree to a 2D texture. This texture is used by the fragment shaders to access the BVH. Table 3.1 and Table 3.2 show how the bounding boxes and triangles are stored in the texture. In addition to the triangle vertices, each leaf node also holds the normal information for each vertex of the triangle. These normals are used for computing the interpolated normal of the point which the ray intersects.

Table 3.1: Data Structure for Bounding Box

R	boundingBox.min.x	boundingBox.max.x
G	boundingBox.min.y	boundingBox.max.y
B	boundingBox.min.z	boundingBox.max.z
A	nodeType	escapeIndex

Table 3.2:Data Structure for Triangle

R	v1.x	v2.x	v3.x	n1.x	n2.x
G	v1.y	v2.y	v3.y	n1.y	n2.y
B	v1.z	v2.z	v3.z	n1.z	n2.z
A	nodeType	materialIndex	n3.x	n3.y	n3.z

Although the preferred texture is 2D, it is considered as a one dimensional array. All addressing is done with a single index which is converted to 2D texture coordinates inside the fragment shader. The *escape indices* for triangles are not stored in the texture since they are found by incrementing the *sequence index* by one. Note that nodes don't necessarily have the same size. This is accounted for during the construction of the tree and when the *escape indices* are set afterwards. Thus, the indices in practice are not consecutive as shown in Figure 3.1 but are incremented by two for bounding boxes and five for triangles. The *nodeType* in both types of nodes is used for deciding whether to test the ray against a bounding box or a triangle. The *materialIndex* field is a pointer to another texture that holds the properties of materials present in the scene.

Algorithm 5 shows the pseudo-code for ray tracing on GPU. Ray directions are calculated on the GPU by interpolating a rectangle that defines the viewing plane. Ray origins are calculated by clipping the rays to the bounding box of the scene using ray directions. The results are written to two textures that are mapped one-to-one to the pixels on the

Algorithm 4 Setting the Escape Indices

```
function SetEscapeIndex(index)
    $escapeIndex \leftarrow$ index
    if leftChild exists then
        leftChild.SetEscapeIndex(rightChild.sequenceIndex)
    end if
    if rightChild exists then
        rightChild.SetEscapeIndex(escapeIndex)
    end if
```

Algorithm 5 Render

```
function RayTrace
    Create Origin and Direction Rays
    repeat
        if First Time then
            Update Depth Buffer using Bounding Box intersection
        else
            Update Depth Buffer using the results of last traversal
        end if
        Run Traverse Kernel
        rendered fragments ← count number of pixels updated
    until rendered fragments = 0
    for each light source do
        Run Shading Kernel
    end for
```

screen, one for ray origins and one for ray directions. These textures hold the origin and direction for each ray in the *RGB* components of the corresponding pixel. The A component is unused in the direction texture. In the origin texture, it is used for marking the pixels that don't intersect the bounding box of the scene with -1. This field is used when the depth buffer is initialized for the first time in the main loop.

During the first iteration, the depth buffer is initialized by marking the pixels that intersect the bounding box of the scene with 0.75 and the pixels that don't intersect the bounding box with 0.25 (the pixels that have -1 in the A component of the ray origin texture). The main kernel which computes the ray-scene intersections is called next. By using early fragment culling, it is guaranteed that the kernel is invoked for only the pixels that have 0.75 in the corresponding pixel of the depth buffer. To exploit this feature, a loop variable is used to control the number of nodes that are tested per pixel in one pass over the GPU Pipeline. Setting this parameter low causes too much API overhead and setting it high causes precious GPU time to be wasted on already finished pixels. An optimal value should be found for increasing performance. Introducing such a variable results in the need for making multiple passes over the main kernel for a single frame. Starting with the second iteration, each pass over the GPU Pipeline updates the depth buffer first. This is needed because the purpose of early fragment culling is to reject finished pixels before the fragment shader is invoked. To do this, finished pixels are marked with 0.25 while unfinished pixels are kept unchanged in the depth buffer.

To extract all necessary information, the main kernel outputs to multiple render targets. Table 3.3 shows the elements of two textures that are output from the main kernel. These two textures are also used as inputs during the multiple passes. Best Hit texture is initialized to $(0.0f, 0.0f, INF, 0.0f)$ where INF is a sufficiently large value, while Render State texture is initialized to $(0.0f, 1.0f, 1.0f, 1.0f)$ before the actual kernel is run. The B component of Best Hit keeps the distance to the current closest triangle that the corresponding ray has intersected. The R component of Render State keeps a pointer to the current node of the tree that needs to be processed. Hence, it is initialized to $0.0f$ since each ray starts the traversal from the root node. The Best Hit and Render State textures can be seen in 3.2. Note that the bounding box can be seen in the Render State texture.

Table 3.3: Best Hit and Render State texture elements

	Best Hit	**Render State**
R	1^{st} barycentric coordinate	next node
G	2^{nd} barycentric coordinate	normal.x
B	distance to hit	normal.y
A	material index	normal.z

For making only the necessary number of passes and not more, a mechanism is needed for deciding when the whole screen is rendered. To do this, Occlusion Querying is used. An occlusion query counts the number of pixels that have their value changed after a fragment shader is run. This count is greater than zero for all passes but one. When it is zero, it is known that the intersection tests have been completed for the whole screen.

After computing the intersections, a shading kernel is run for each light source to get the final image.

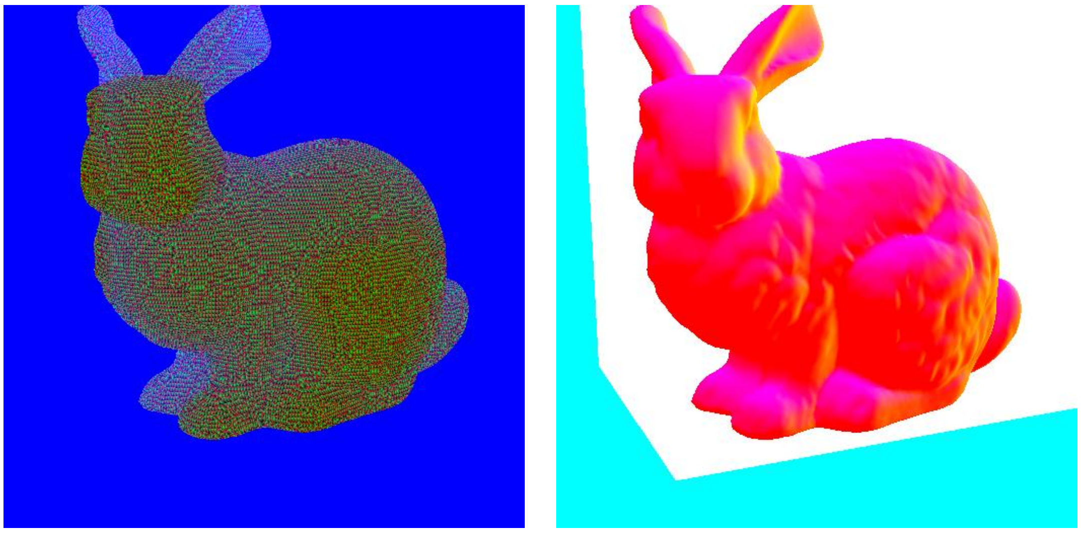

(a) **(b)**

Figure 3.2: Main kernel outputs for Stanford Bunny **(a)** Best Hit **(b)** Render State

3.3 Parallelization

To parallelize ray tracing over the GPU cluster that is explained in Section3.2, a *task parallel* Client/Server model is chosen. The reason for this is to designate one computer in the cluster to interact with the user and display the rendered frames. There are two processes that share the role of the Server. These are called Display and Master. All other processes in this parallelization are called Slaves. The roles of these processes will be explained shortly. Figure 3.3 shows the placement of these processes in the cluster.

Algorithms 6, 7 and 8 demonstrate the work done by Display, Master and Slaves, respectively. Blocking calls are used throughout the implementation. Display is responsible for opening up a window and displaying the final image. The user interacts with this process. Display keeps the camera data and is also responsible for delivering that data to all slaves before each frame is rendered. The screen is divided into a number of unit tiles. The division of the screen can be seen in Figure 3.4. These tiles are sent to slaves as they become available starting from the top left corner. Throughout the rendering process, when a slave is done with the rendering of a tile, the subimage that corresponds to that tile is sent to the Display node. At this point, the Slave waits for the Master to send a new tile. If there are tiles that haven't been rendered yet, the Master orders the Slave to render the next tile. This process goes on until all tiles are rendered, at which point, the Slaves are sent the DONE signal. The Display goes back to waiting for input state while the Master and Slaves go

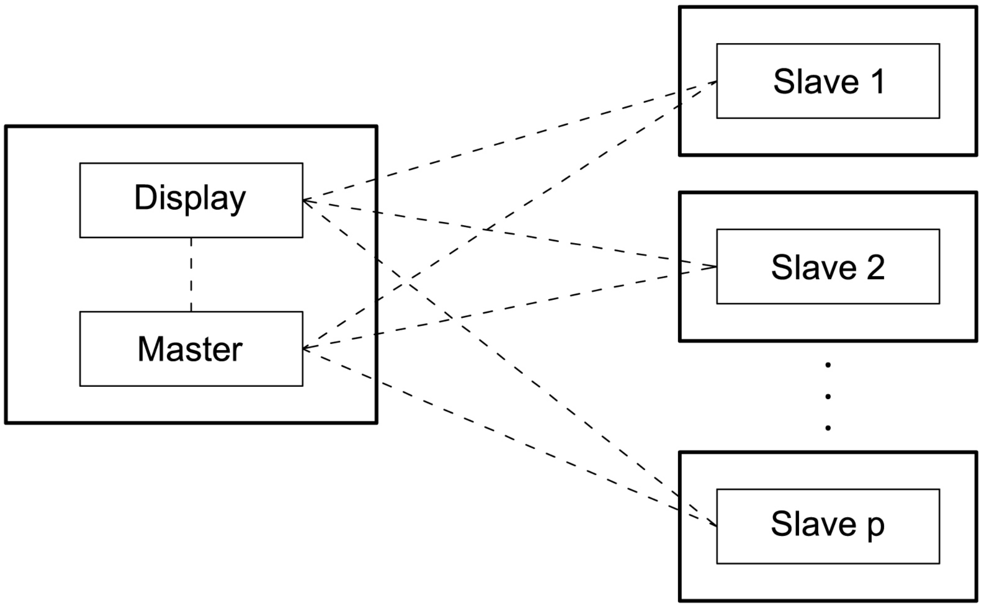


Figure 3.3: The layout of the cluster

back to the initial state of waiting for a new frame.

The reason for dividing the role of server to two processes is the lack of multi-thread support in the MPI implementation that is used. Also, not to introduce an overhead by querying statuses of nonblocking calls, all message passing is done using blocking calls. Thus, Display and Master processes are run on the same computer and simulate a multi-threaded node in the cluster. The messages passed between the two processes are limited to two integers per frame, which means that the communication overhead is negligible when the total frame time is considered.

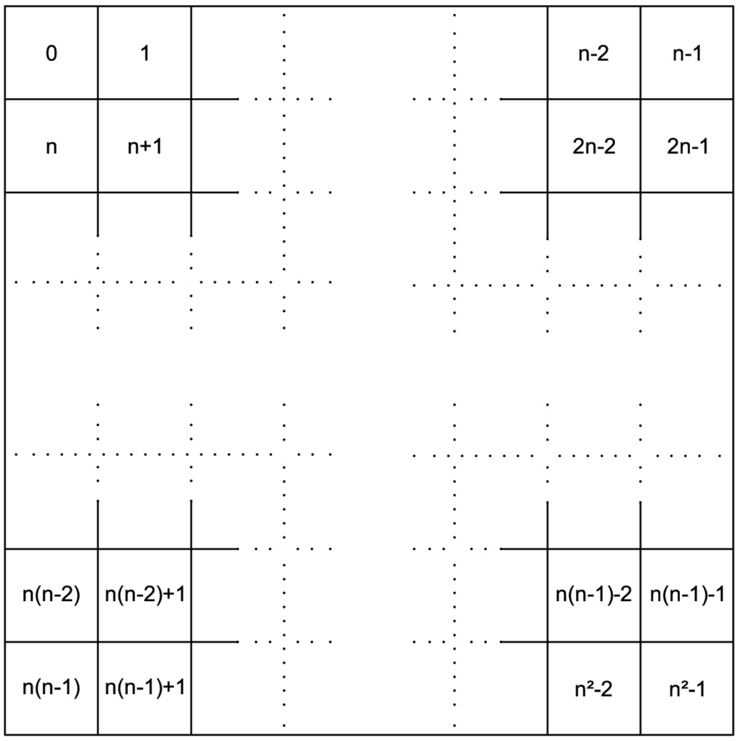


Figure 3.4: The division of the screen into tiles

Algorithm 6 Display Pseudo-code

```
function RayTrace
    Send RENDER to Master
    Send RENDER to Slaves
    Send CAMERA to Slaves
    while there are tiles to be rendered do
        Receive a rendered tile from SLAVE
    end while
    Display final image
```

Algorithm 7 Master Pseudo-code

```
function RayTrace
    loop
        Receive command from Display
        if command = QUIT then
            break
        end if
        $currentTile \leftarrow 0$
        while there are tiles to be rendered do
            Receive READY from ANY slave
            Send $currentTile$ to slave
            $currentTile++$
        end while
    end loop
```

Algorithm 8 Slave Pseudo-code

```
function RayTrace
    loop
        Receive command from Display
        if command = QUIT then
            break
        end if
        Receive CAMERA from Display
        loop
            Send READY to Master
            Receive command from Master
            if command = DONE then
                break
            end if
            $currentTile \leftarrow command$
            Render $currentTile$
            Send $currentTile$ to Display
        end loop
    end loop
```

3.4 Summary

The main work of this thesis has been explained in this chapter. Task parallelism is used for distributing the work between processes, each of which has access to the whole scene and use BVH to decrease the number of ray-triangle intersection tests. Next chapter presents the results and interprets them in the context of this thesis.

CHAPTER 4

EXPERIMENTS

In this chapter, the experimental setup that is used in this study will be explained. After doing that, the results of various experiments will be shared with relevant discussion.

4.1 Experimental Setup

The experiments are performed on a cluster of five computers. Four of these computers are identical and consist of Intel Pentium 4 3.20 GHz processors with 1 GB of RAM and GeForce 7800 GTX graphical processors. These computers are used for running the Slave processes. The fifth one consists of an Intel Core2Duo 2.00 GHz processor with 2GB of RAM and GeForce Go 7900 GS graphics processor. This computer is used for running the Master and Display processes. The computers are connected with a Gigabit Ethernet Switch. For development, C++ and Cg is used. For implementing the intra-cluster communication, MPICH 1.2.5 [27] is employed. The models that are used during the tests are Stanford Bunny (69451 triangles), teapot (16128 triangles), sphere flake (88562 triangles), lattice (125388 triangles) and town (78028 triangles).

In the serial implementation, the whole screen is rendered in one quad. The find the best loop parameter, tests are done by setting the loop parameter to 8, 16, 32, 64 and 128. The camera is rotated around the screen and is placed in intervals of 10 degrees for a total of 36 camera angles. All settings are tested 10 times and the best result is stored.

For parallel implementation, the loop parameter that gave the fastest render time during the serial tests is used which is 16 for all scenes. The number of tiles used for testing are 4, 16, 64, 256 and 1024. Tests with 2, 3 and 4 processes are conducted. All tests are done for 256×256, 512×512 and 1024×1024 screen resolutions. The camera was positioned at the best and worst angles which were found during the serial tests.

4.2 Results

The purpose of the first group of tests was to determine the best and worst angles of the BVH. The results can be seen in Figure 4.1. For each scene, there is a continuous range of camera positions for which the rendering takes significantly longer. The reason for this difference is the fixed-order BVH traversal that is mentioned in Chapter 3. The best angle that is found is the angle where minimal number of ray-triangle intersection tests are needed. The worst angle, on the other hand, requires making significantly more number of tests compared to the best angle. Depending on the direction of the rays, distinct BVHs perform differently. The rest of the tests were carried out for the best and worst angles to get an understanding of how the cluster performs with varying computational intensities.

The next portion of the tests were done for deciding the number of tiles per resolution for each scene. Figure 4.2 and Figure 4.3 show the results of these tests for the best and worst BVH angles respectively. All resolutions and scenes performed best with 16 tiles except three setups. These are Stanford Bunny - 256×256 (4 tiles), Lattice - 512×512 (64 tiles) and Lattice - 1024×1024 (64 tiles), all with the worst BVH angle. These figures show that at low resolutions, performance drops faster when the number of tiles is increased. This is due to the drop in the computation-to-communication ratio. Even though this drop is present at each resolution, the computations that are done for lower resolutions are not as intense as those at higher resolutions. For smaller number of tiles, all resolutions perform very close. As the number of tiles are increased, speedup drops a lot faster for low resolutions while the drop in high resolutions aren't as steep. Moreover, the ratio of speedup at high resolutions to speedup at low resolutions is very close for both angles.

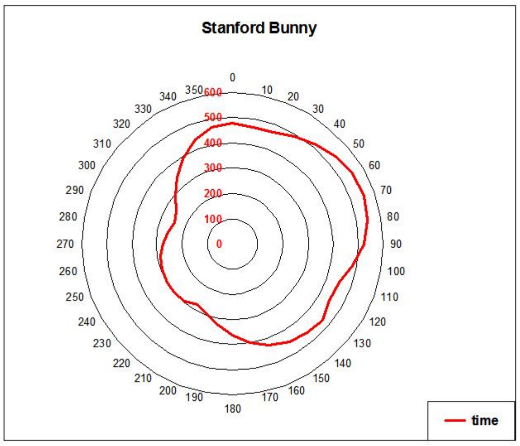
Figure 4.4, Figure 4.5 and Figure 4.6 show the speedup and efficiency for the three resolutions and the two angles. Other than the Lattice - 256×256, tests that use the worst BVH angle yield better results than the tests that use the best BVH angle. Again, this is the result of the drop in the computation-to-communication ratio. Another statistic that supports this trend is this: for each scene, high resolutions almost always yield better results compared to lower resolutions. The sphere flake, which was the slowest scene in the serial tests, had a speedup of 3.90 at 1024×1024 which also supports this effect. Table 4.1, Table 4.2 and Table 4.3 show the speedup and efficiency for each resolution and the two angles. 256×256 resolution does not take advantage of the parallelism as much as the higher resolutions since the API and network overheads dominate the total time. In fact, using best BVH angle, a two-process setup performs worse than the serial implementation for two scenes at lowest resolution. This can only happen if the overhead introduced by API and network is so big that the speed gained from parallelizing the ray tracing is diminished.

For all resolutions, the speedup of Lattice for worst BVH angle and best BVH angle is almost same. The reason for this is the uniform distribution of triangles in the scene.

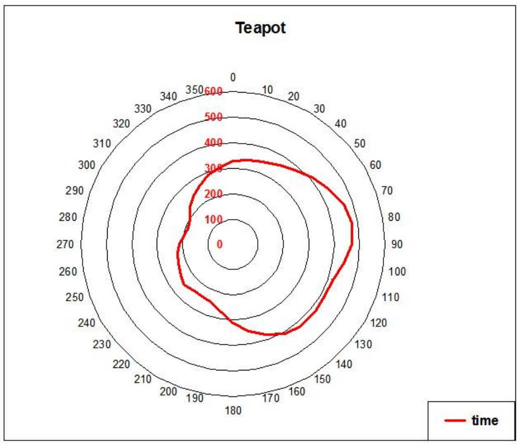
Figure 4.7 shows the overhead per pixel that is introduced by API and communication calls. To determine the API overhead, serial implementation was used with the listed number of tiles for each scene using the best angle. The total time to render the scenes increased significantly. The network overhead was found by using three processes, one Display, one Master and one Slave, again with the listed number of tiles and comparing these timings with the results of the first group of serial experiments. While the first group of experiments that is mentioned gives accurate values, the second group of experiments don't include the idle durations that occur when slaves have to wait for the Display to receive previous tiles. Results show that API overhead is about twice as much as the network overhead. Also, the overhead per pixel at 256×256 resolution is about nine times that of 1024×1024. This suggests that, theoretically, small number of tiles at high resolutions decrease the total overhead per pixel. However, the overall overhead for two settings isn't proportional to the number of pixels in a tile. For example, although 1024 tiles introduce the least overhead per pixel, performance is significantly lower than 256 tiles. Results show that 16 tiles is critical where the combined effect of API and network overheads and idle durations is low enough to allow the cluster to achieve maximum performance.

Throughout the experiments, the ratio of time actually spent on either computation or API calls to total rendering time ranged from 85% for sphere flake (worst BVH, 1024×1024) to 50% for teapot (best BVH, 256×256). This observation is not surprising because the mentioned sphere flake takes longest to render while the mentioned teapot requires the least amount of time using the serial implementation. This is consistent with the previous comments about the effect of computational intensity on speedup.

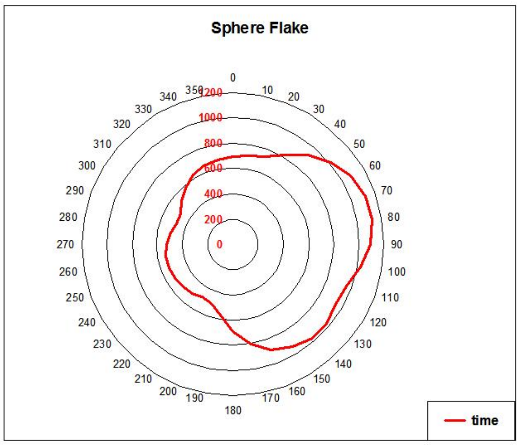
Figure 4.8 through Figure 4.12 show the ray traced images produced by the cluster.



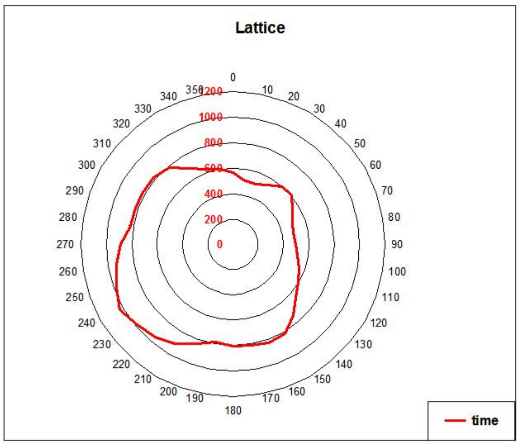
(a)



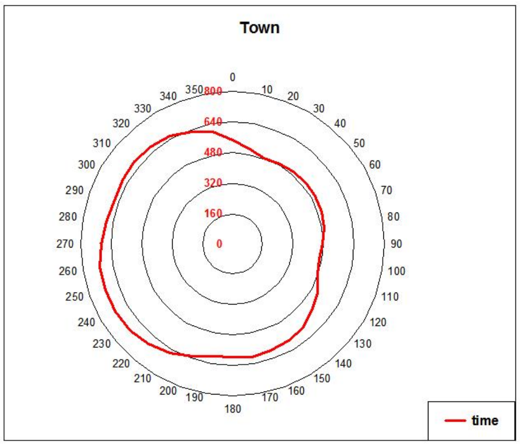
(b)



(c)

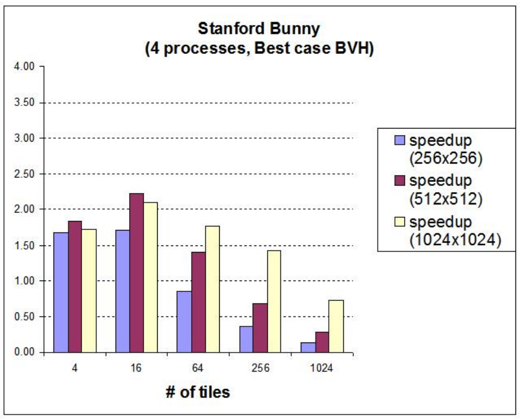


(d)

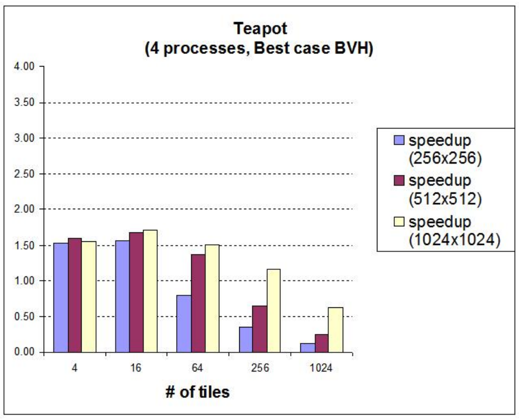


(e)

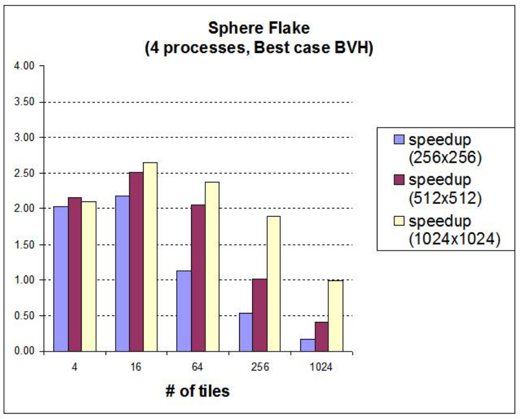
Figure 4.1: Render Time vs Camera Angle. **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town



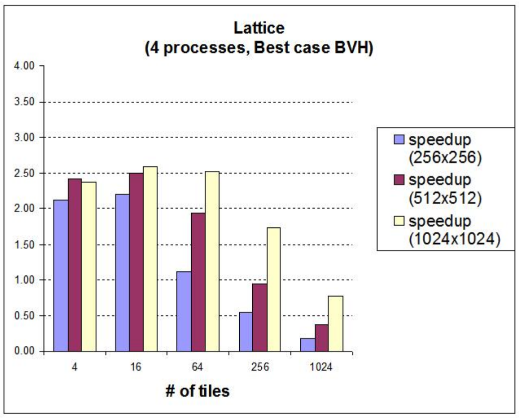
(a)



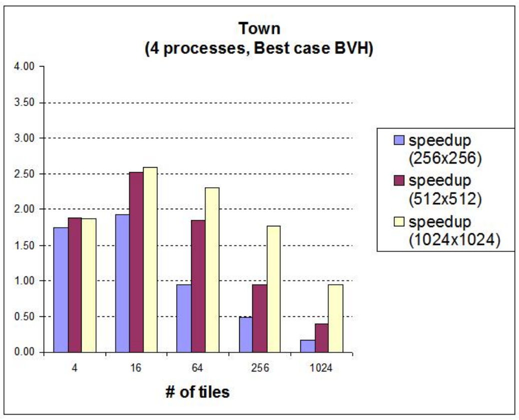
(b)



(c)

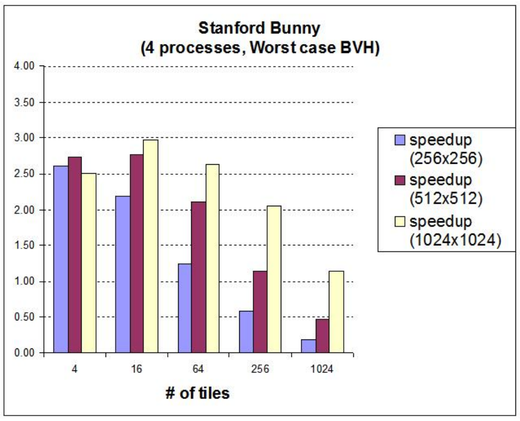


(d)

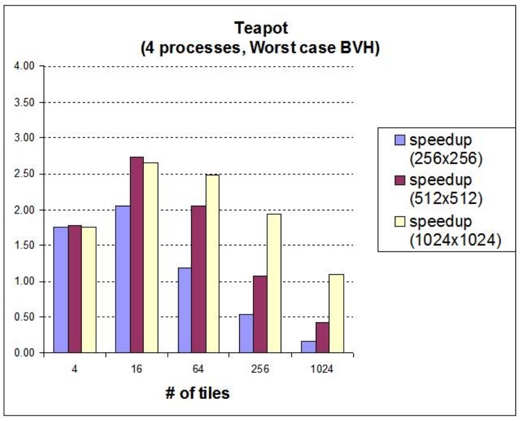


(e)

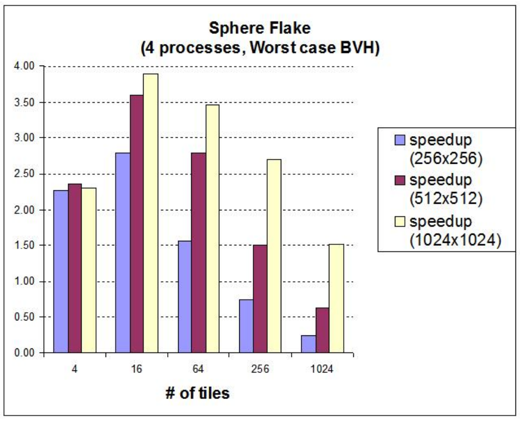
Figure 4.2: Speedup vs Number of Tiles from the best angle for all scenes. **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town



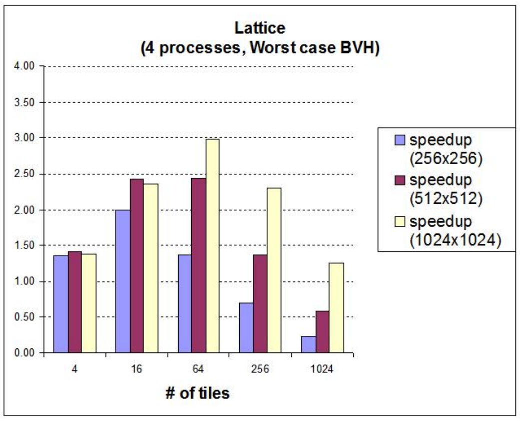
(a)



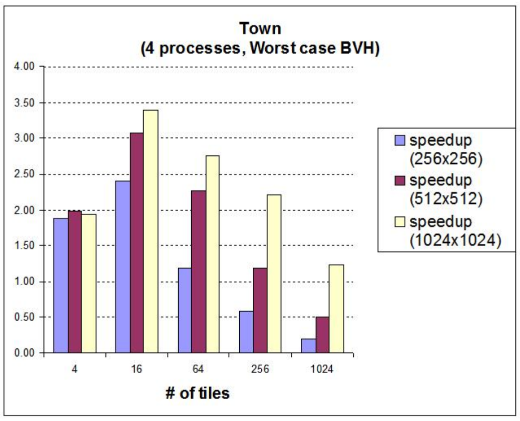
(b)



(c)

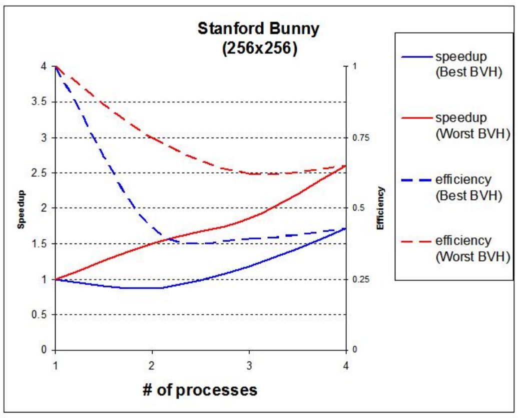


(d)

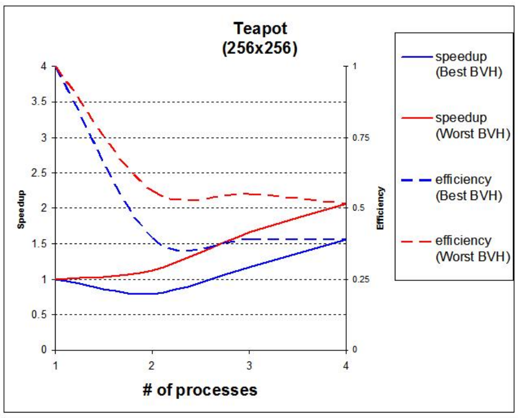


(e)

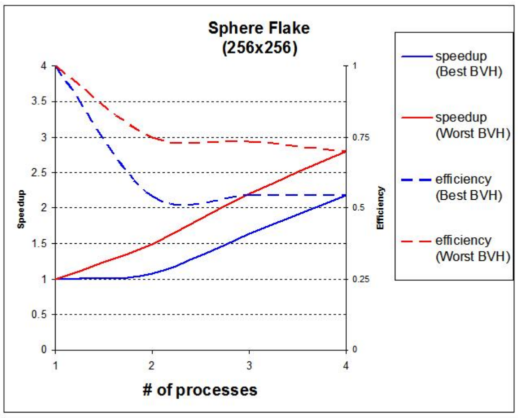
Figure 4.3: Speedup vs Number of Tiles from the worst angle for all scenes. **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town



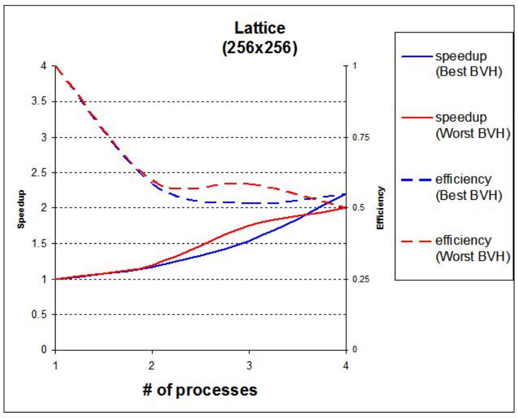
(a)



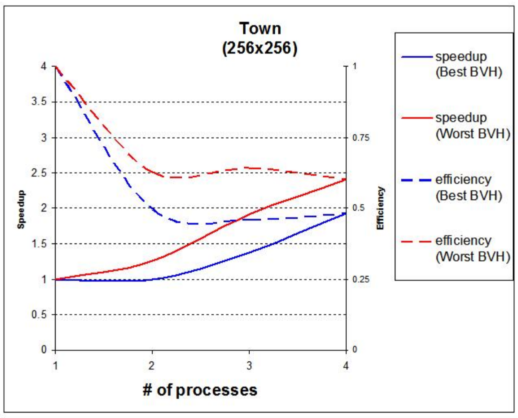
(b)



(c)

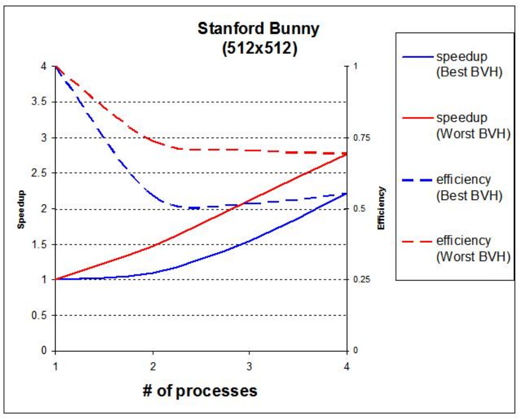


(d)

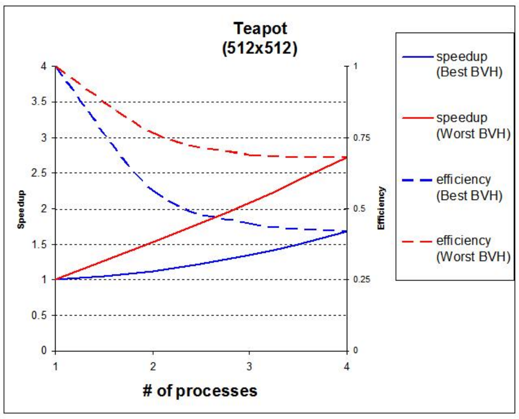


(e)

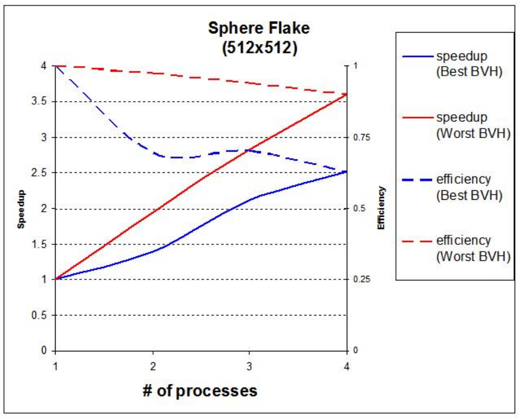
Figure 4.4: Speedup and Efficiency vs Number of Tiles (256×256). **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town



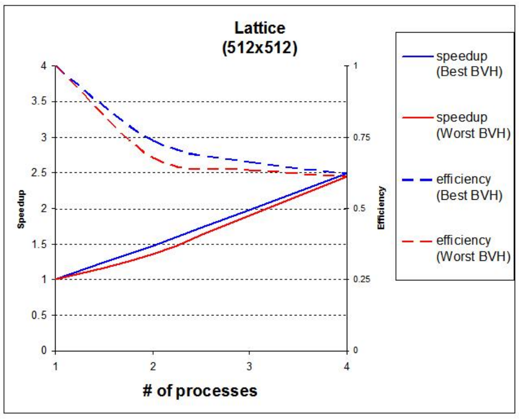
(a)



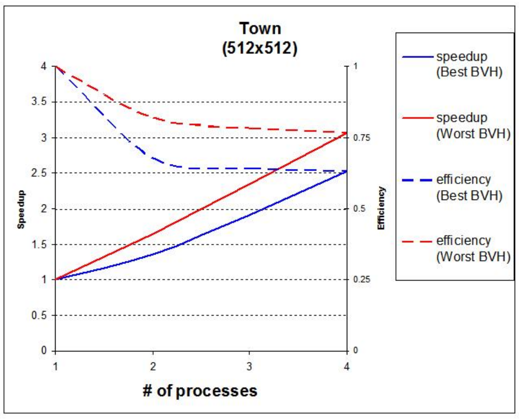
(b)



(c)

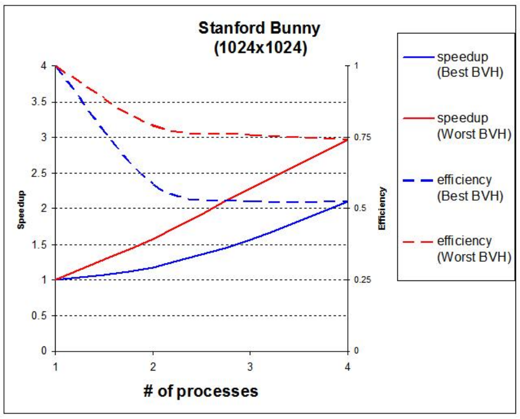


(d)

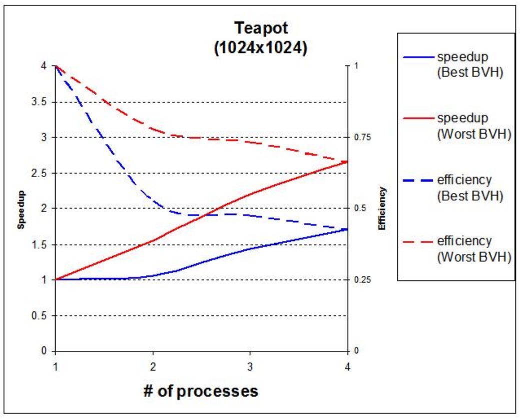


(e)

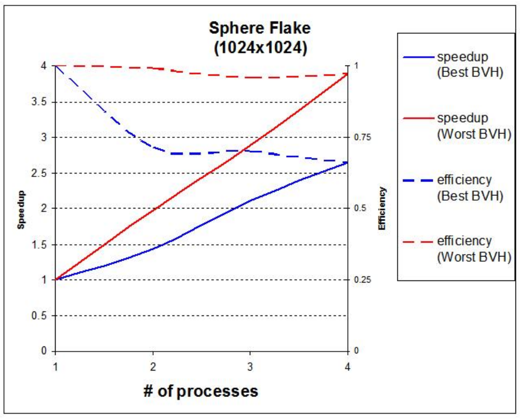
Figure 4.5: Speedup and Efficiency vs Number of Tiles (512×512). **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town



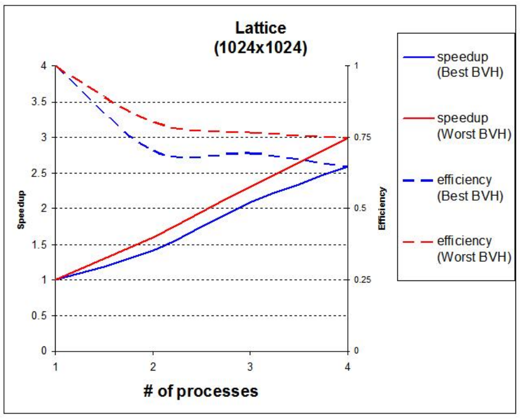
(a)



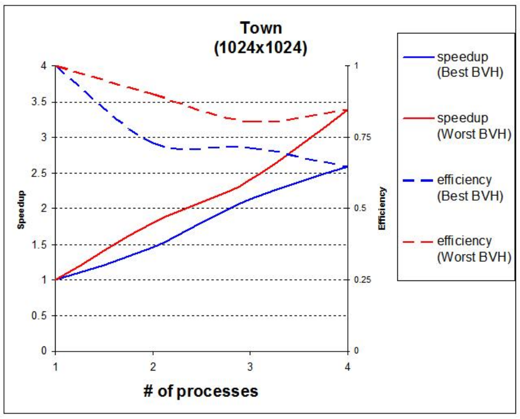
(b)



(c)



(d)



(e)

Figure 4.6: Speedup and Efficiency vs Number of Tiles (1024×1024). **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town

Table 4.1: Speedup and Efficiency (256×256)

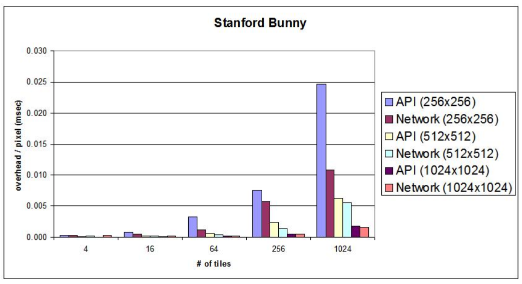
	Best BVH		Worst BVH	
	Speedup	Efficiency	Speedup	Efficiency
Stanford Bunny	1.71	%42.8	2.60	%65.1
Teapot	1.56	%39.1	2.06	%51.4
Sphere Flake	2.18	%54.5	2.80	%70.0
Lattice	2.20	%55.0	2.00	%50.0
Town	1.93	%48.3	2.40	%60.0

Table 4.2: Speedup and Efficiency (512×512)

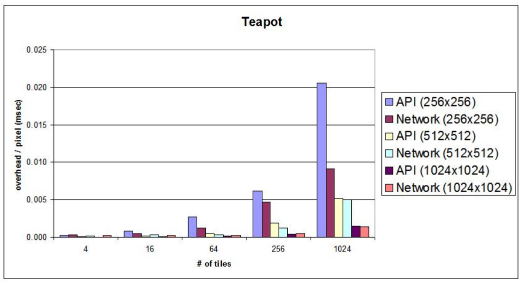
	Best BVH		Worst BVH	
	Speedup	Efficiency	Speedup	Efficiency
Stanford Bunny	2.22	%55.5	2.77	%69.3
Teapot	1.67	%41.7	2.73	%68.2
Sphere Flake	2.51	%62.8	3.60	%90.0
Lattice	2.49	%62.2	2.44	%61.1
Town	2.52	%63.0	3.07	%76.7

Table 4.3: Speedup and Efficiency (1024×1024)

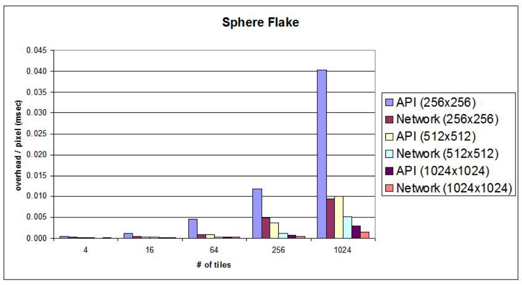
	Best BVH		Worst BVH	
	Speedup	Efficiency	Speedup	Efficiency
Stanford Bunny	2.09	%52.3	2.97	%74.2
Teapot	1.71	%42.6	2.66	%66.4
Sphere Flake	2.65	%66.2	3.90	%97.5
Lattice	2.59	%64.8	2.98	%74.6
Town	2.59	%64.8	3.39	%84.8



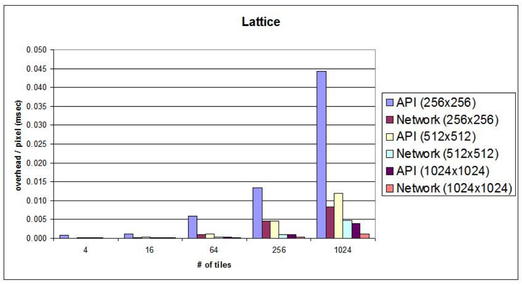
(a)



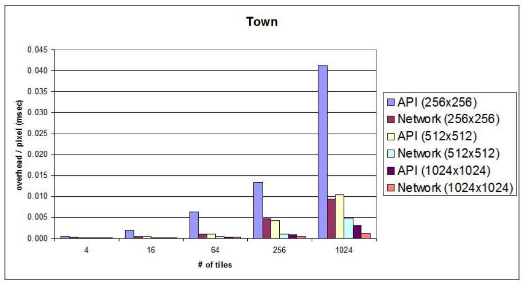
(b)



(c)



(d)



(e)

Figure 4.7: API and Network Overhead vs Number of Tiles for all resolutions. **(a)** Stanford Bunny **(b)** Teapot **(c)** Sphere Flake **(d)** Lattice **(e)** Town

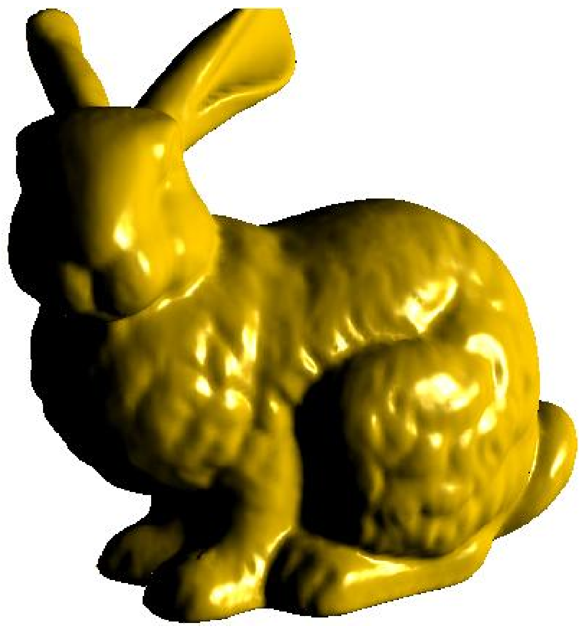


Figure 4.8: Stanford Bunny

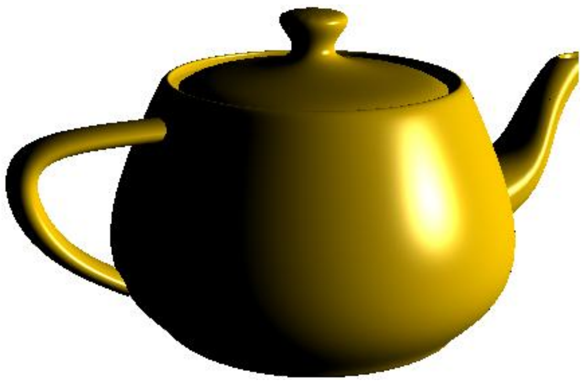


Figure 4.9: Teapot

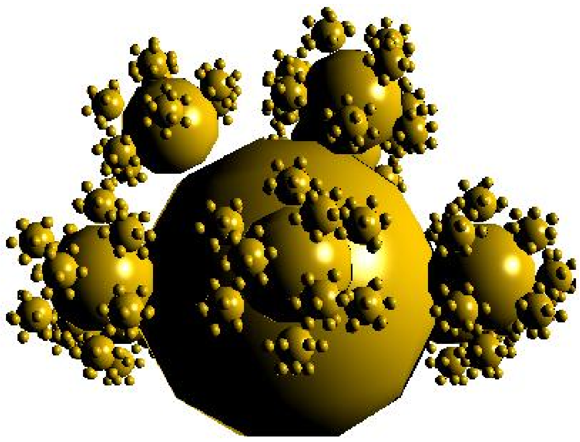


Figure 4.10: Sphere Flake

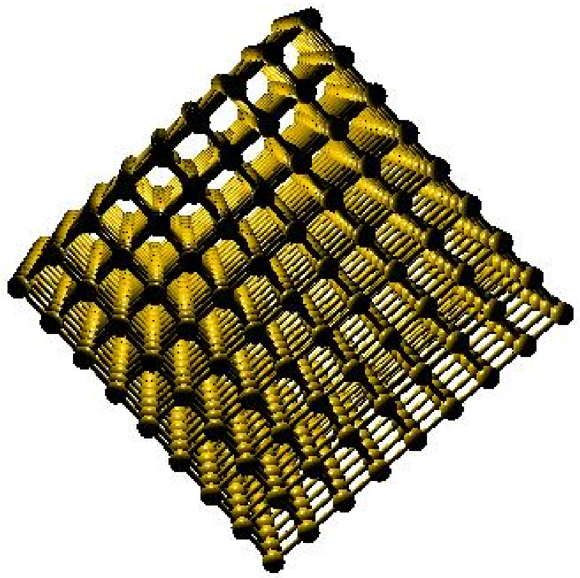


Figure 4.11: Lattice

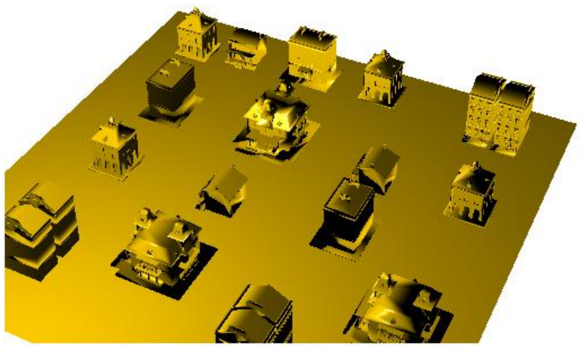


Figure 4.12: Town

CHAPTER 5

CONCLUSION

In this thesis, task parallel ray tracing on a GPU cluster is presented. For the GPU implementation, the BVH traversal algorithm proposed by Thrane & Simonsen [38] is used because of its performance and simplicity. A task parallel approach is employed to parallelize ray tracing on the cluster where unit tiles are rendered by slave processes. When a process is done with a tile, it requests a new tile from the Master until the whole screen is rendered.

The results are not surprising. Difficult scenes at high resolutions exploit parallelism much better than easier scenes at lower resolutions. The BVH implementation used in this thesis causes varying timings for the serial tests which is reflected in the parallel tests too, with worst BVH angle speedups being consistently higher than the best BVH angle speedups. The sphere flake which was the hardest scene to render with a single process reached an efficiency of 97.5% using the worst BVH angle at highest resolution.

Although increasing the number of tiles helps load balancing, it introduces too much overhead. 16 was the setup that yielded the highest speedup in almost all settings. Using more than 64 tiles causes performance to decrease consistently.

Processes were kept busy when difficult scenes were rendered at high resolutions. However, API and network overheads dominated total time in easy scenes, especially at low resolutions.

The biggest obstacle for scalability is the network and API overheads. Another factor that causes problems for scalability is the decomposition strategy that is used. Task parallel ray tracing can only render scenes that fit in the memory that is accessible by each process. For bigger scenes, data parallelism is a must.

The most important observation so far is that high computational intensity exploits parallelism better. Our insights indicate that better speedups can be achieved by ray tracing bigger scenes, or increasing the resolution and keeping the number of tiles same.

Small number of tiles introduce unwanted idle durations to slaves that finish early, but the rate at which the API overhead increases is not encouraging when higher number of tiles is considered for better load balancing. Thus, a method is needed to decrease the number of tiles while still having efficient load balancing. Currently, the size of the tiles are constant throughout the execution. There is no way to change the size depending on the characteristics of the scene. Therefore, even if a big part of the screen doesn't intersect the scene, artificial API and network overheads are introduced by rendering that area in multiple fixed-sized tiles. This overhead takes a bigger portion of the total time as the resolution decreases. As a future work, it may be possible to increase performance by assigning a bigger tile to a process when a group of pixels are known to be not intersecting the scene. To keep the slave processes busy at the end, the master process could send tiles to multiple processes and use the tile from the process that finishes first.

In order to decrease API overhead at the beginning and decrease idle duration at the end of a frame, one could start with big tiles and gradually decrease the tile size to increase performance.

Having computers with different computational capacities could very easily lead to random frame times with high variance and result in a huge performance drop. Therefore, distinguishing between the capabilities of computers and assigning harder tasks to more powerful ones seems like a must for bigger clusters. The master process could assign tiles to slave processes depending on past performance. This would make using different computers with varying performances possible.

Although the network overhead isn't as much as the API overhead, it should be decreased to improve performance. In the current implementation, two processes can finish their tiles at the same time and one of them may have to wait until the other successfully delivers its rendered tile to the Display process. Note that this occurs more frequently with bigger number of tiles. To do this, the Display node may have separate threads for each slave which would avoid idle duration in the mentioned case. The slaves could also be multi-threaded for rendering new tiles and sending finished tiles concurrently.

Another way of increasing performance may be to send coherent groups of tiles to individual processes instead of a first-come, first-served basis that is demonstrated in Chapter 3. This way, processes would be working on related pixels and GPU cache would be utilized for faster computation.

REFERENCES

[1] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. *Eurographics '87*, pages 3–10, 1987.

[2] AMD. AMD "Close to Metal" Technology Unleashes the Power of Stream Computing. *http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543_544~11-4147,00.html*, Last visited December 2007.

[3] Arthur Appel. On calculating the illusion of reality. In *IFIP Congress (2)*, pages 945–950, 1968.

[4] Nathan A. Carr, Jesse D. Hall, and John C. Hart. The ray engine. *Graphics Hardware (2002)*, pages 1–10, 2002.

[5] Alan Chalmers, Timothy Davis, and Erik Reinhard. *Practical Parallel Rendering*. AK Peters, 2002.

[6] Daniel Cohen and Zvi Sheffer. Proximity clouds - an acceleration technique for 3d grid traversal. *Vis. Comput.*, 11(1):27–38, 1994.

[7] Ş. Alphan Es. *Accelerated Ray Tracing Using Programmable Graphics Pipelines*. Ph.d. thesis, Middle East Technical University, January 2008.

[8] Jack Dongarra, Ian Foster, Geoffrey C. Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, editors. *The Sourcebook of Parallel Computing*. Morgan Kaufmann, November 2002.

[9] Manfred Ernst, Christian Vogelgsang, and Günther Greiner. Stack implementation on programmable graphics hardware. In *VMV*, pages 255–262, 2004.

[10] Alphan Es and Veysi İşler. Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor. *Journal of Parallel and Distributed Computing*, 67(11):1201–1217, 2007.

[11] Zhe Fan, Feng Qui, Arie Kafuman, and Suzanne Yoakum-Stover. Gpu cluster for high performance computing. 2004.

[12] Tim Foley and Jeremy Sugerman. Kd-tree acceleration structures for a gpu raytracer. *IEEE Computer Graphics and Applications*, 7(5):14–20, 2005.

[13] Dominik Göddeke. GPGPU::Basic Math Tutorial. *http://www.mathematik.uni-dortmund.de/ goeddeke/gpgpu/tutorial.html#feedback2*, Last visited November 2007.

[14] Andrew S. Glassner. *An Introduction to Ray Tracing*. Morgan Kaufmann, January 1989.

[15] Jeffrey Goldsmith and John Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[16] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[17] RapidMind Inc. Sh: A high-level metaprogramming language for modern GPUs. *http://libsh.org/*, Last visited December 2007.

[18] Intel. Forthcoming Dual-Core Intel® Itanium® Processor Achieves Fastest Four-Way Floating Point Benchmark. *http://www.intel.com/pressroom/archive/releases/-20050707corp.htm*, Last visited December 2007.

[19] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. AK Peters, July 2001.

[20] Timothy L. Kay and James T. Kajiya. Ray tracing complex scenes. *ACM Computer Graphics*, 20(4):269–278, 1998.

[21] Stanford University Graphics Lab. BrookGPU. *http://graphics.stanford.edu/projects/-brookgpu/index.html*, Last visited December 2007.

[22] Stephen Lacey and Richard Box. A fast, easy sort. *BYTE*, 16(4):315–ff., 1991.

[23] LLNL. Introduction to Parallel Computing. *https://computing.llnl.gov/tutorials/parallel-_comp/*, Last visited December 2007.

[24] David Luebke and Mark Harris. Gpgpu: General purpose computation on graphics hardware. Technical report, SIGGRAPH, 2005.

[25] Microsoft. DirectX. *http://www.gamesforwindows.com/en-US/AboutGFW/Pages/Direct-X10.aspx*, Last visited December 2007.

[26] Microsoft. HLSL Workshop. *http://msdn2.microsoft.com/en-us/library/bb173495-(VS.85).aspx*, Last visited December 2007.

[27] MPI. Message Passing Interface. *http://www-unix.mcs.anl.gov/mpi/*, Last visited October 2007.

[28] nVidia. CUDA. *http://developer.nvidia.com/object/cuda.html*, Last visited November 2007.

[29] nVidia. Cg. *http://developer.nvidia.com/page/cg_main.html*, Last visited October 2007.

[30] OpenGL. OpenGL Shading Language. *http://www.opengl.org/documentation/glsl/*, Last visited December 2007.

[31] OpenGL. WGL_ARB_pbuffer. *http://www.opengl.org/registry/specs/ARB/wgl_pbuffer.txt*, Last visited December 2007.

[32] OpenGL. OpenGL - The Industry Standard for High Performance Graphics. *http://www.opengl.org/*, Last visited October 2007.

[33] POV-Ray. The Persistence of Vision Raytracer. *http://www.povray.org/*, Last visited December 2007.

[34] Timothy John Purcell. Ray tracing on a stream processor. 2004.

[35] PVM. Parallel Virtual Machine (PVM) Version 3. *http://www.netlib.org/pvm3/*, Last visited October 2007.

[36] SGI. EXT_framebuffer_object. *http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt*, Last visited October 2007.

[37] Milos Sramek and Arie Kaufman. Fast ray-tracing of rectilinear volume data using distance transforms. *IEEE Transactions on Visualization and Computer Graphics*, 06(3):236–252, Jul-Sept 2000.

[38] Niels Thrane and Lars Ole Simonsen. *A Comparison of Acceleration Structures for GPU Assisted Ray Tracing*. Master's thesis, University of Aarhus, August 2005.

[39] Ingo Wald, Carsten Benthin, Andreas Dietrich, and Philipp Slusallek. Interactive Distributed Ray Tracing on Commodity PC Clusters – State of the Art and Practical Applications. *Lecture Notes on Computer Science*, 2790:499–508, 2003. (Proceedings of EuroPar 2003).

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

[41] Wikipedia. GeForce 8 Series. *http://en.wikipedia.org/wiki/GeForce_8_Series*, Last visited November 2007.

[42] Wikipedia. Parallel Computing. *http://en.wikipedia.org/wiki/Parallel_computing*, Last visited November 2007.