Real-time Ray Tracing using CUDA

Michael Allgyer

4/12/2008

Master's Project Report

Signatures:

Advisor (Joe Geigel): ____

Reader (Warren Carithers): ____

Graduate Advisor (Hans-Peter Bischof): ________________________________

Table of Contents:

Section 1: Abstract

Ray tracing is a widely used and well-studied algorithm that produces high-quality computer generated images. However, the algorithm requires enormous amounts of computation, and as a result cannot be efficiently done in real-time on commodity hardware. While current graphics processing units (GPUs) use rasterization to render images, graphics company Nvidia has released CUDA, a freely available SDK which allows developers to create C programs that run on CUDA-compatible GPUs. This project investigates CUDA's processing power, parallel hardware, and memory management, and maps it to ray tracing in order to see how it can perform the algorithm in real-time. The result is a fully interactive ray tracing system that utilizes a GPU's parallel architecture.

Section 2: Statement of Problem

One of the main goals of computer graphics is to make a computer generate photorealistic images from scene data. Numerous techniques have been developed to accomplish this, including rasterization, ray tracing, and radiosity. All three of these techniques have advantages and disadvantages. For example, current commercial hardware can do rasterization very quickly and efficiently, but ray tracing usually produces higher-quality images, albeit much more slowly. In fact, for the most part, ray tracing cannot be done in real-time on common computers. This project aims to fix this by developing a real-time ray tracer on commercial hardware.

Section 3: Analysis

Ray tracing is a fairly well-studied process. It generates an image by spawning one or more rays per pixel into the world, testing intersection between the ray(s) and objects in the scene, and coloring the given pixel based on which objects are hit. Considering a common screen resolution is 1280x1024, one image can easily contain more than one million pixels. Then, an average scene in a computer game can contain more than one hundred thousand triangles. This means a naïve ray tracer could need to perform a trillion intersection tests in order to produce a single image. Furthermore, an acceptable frame rate for games is about 60 frames per second, so a real-time ray tracer would need to generate an image in about 1/60th of a second. Considering a single intersection test contains approximately a few dozen floating point operations, the amount of computation power required is quite large; even today's fastest multi-core processor cannot come close to this performance.

Because ray tracing produces such high-quality images, attempting to do it in real time is not a new concept. This field of study could also be simply called optimizing ray tracing, and more or less comes in two flavors: optimizing with hardware or with software.

Performing real-time ray tracing in software usually means optimizing the basic algorithm so fewer calculations need to be performed. One common way to do this is to perform intersection tests only on those objects that are in the viewing volume of the camera. This is done by using one of a number of well-defined spatial subdivision algorithms such as BSP or kd-trees. Binary Space Partitioning transforms a space into a binary tree representation by recursively subdividing it into convex sets. The basic algorithm divides the space in two until a specific condition is met. How the scene is divided depends on the application[1]. Kd-trees are a special type of BSP where the space is divided on the axes into rectangles or cubes[4]. Another common way to limit the number of intersection tests is to use bounding volume hierarchies. Here, objects that are near each other in a scene are encapsulated in a simple shape – spheres and cubes are common choices. Intersection tests are then performed on the bounding volume, then on the contained objects only if the bounding volume intersected. Of course, bounding volumes can be grouped in larger bounding volumes, yielding a hierarchy[9]. It should be noted these optimizations are quite straightforward in static environments, but in scenes where the objects move the algorithms can become more challenging. Sometimes, the trees or hierarchies simply need to be recalculated for each frame, so these representations need to be efficiently calculated as well.

There has been a significant amount of research in improving the problem with dynamic scenes. One example is contained in a dissertation by Ingo Wald. In it, Wald makes an observation that objects' behaviors can be classified into three categories: static, hierarchical motion, and unstructured motion. Static objects do not change in a scene, hierarchical motion divides objects into hierarchies that are transformed uniformly, and unstructured motion moves each vertex or triangle independently of all other vertices. For static objects, the usual, highly-optimized kd-tree or BVH can be created one time and be used throughout the application. When using hierarchical motion, the ray tracer can transform the rays instead of the objects, thus keeping a static hierarchy or tree. Last, unstructured motion does require modifying the tree, but the paper proposes optimized algorithms to performing this restructuring. One other note worth mentioning is the ray tracer does not decide which class an object should be in; rather the client application tells the ray tracer how to handle each object.

To deal with unstructured motion, Wald notes the tree may need to be recreated every frame. However, this is only necessary when the object's triangles have changed. Also, it can be done only when a ray needs to be tested against the object. Restructuring still takes time though, so Wald compromises structure optimality for build time. For example, he relaxes the subdivision criterion for the BSP. One way he decreases build time is he allows more triangles per node, which gives a shallower tree and faster build time. He notes that scenes usually contain more static and hierarchical motion than unstructured motion, so this system is efficient in most cases. He goes on to describe his top-level kd-tree which handles the scene as a whole[10].

One other approach to optimizing spatial representations in ray tracing is proposed by Ingo Wald, Solomon Boulos, and Peter Shirley. Here, the writers argue that while kd-trees have gotten more attention and have therefore become very efficient, BVHs are more applicable to dynamic (and interactive) scenes. So, the writers implement a ray tracer that uses binary BVHs with axis-aligned bounding boxes. The first half of the paper describes their BVH structure, tree traversal, and usage of packets for single frames. Then, they discuss how BVHs can be used in interactive scenes. First, the topology of the BVH does not need to change between frames. Instead, the dimensions of the volumes can be refit to reflect transformations in the scene. The only question here is how to construct the initial BVH. For this, two options are offered. The first possibility is to create the BVH from the character's first frame in an animation, or its rest pose. However, this could yield a very inefficient BVH (for example if the characters has its hands behind its head in the first frame, one node might contain both its hands and head), so the second option is to create a BVH from a number of valid poses and choose the “best” one from some kind of heuristic[11].

One hardware solution to speed up ray tracing is presented in a paper by Ulf Ochsenfahrt and Ralf Salomon. This paper recognizes the fact the main factor that cripples a ray tracer’s speed is calculating the ray-triangle intersections, and modern software optimizations (regular grids, kd-trees, bounding box hierarchies, etc) cannot guarantee specific improvements in performance under all circumstances. Because of this, the proposed implementation aims to vastly improve that specific point in the ray tracing pipeline. To do this, the writers suggest a hardware design called Constant-time Raytracing with Embedded Memory Architecture, or CREMA. The main idea is it will reduce all ray-object intersections to O(1). This is done with a rather brute-force approach by having a nano processor for every primitive in the scene. All intersection calculations can then be performed simultaneously, thus giving a computational complexity of O(1). This is fundamentally different than the more common ray tracer in which pixels are separated into threads and are computed independently of each other.

This paper describes an implementation the writers produced, albeit on a limited budget, that can be considered successful. Their prototype ran at 13 frames per second with a resolution of 256x128. With better hardware this performance could certainly be improved, but this method inherently imposes a very strict limitation on the maximum number of objects in a scene. While all graphics solutions have limitations, this solution has well-defined limitations, which may be good or bad depending on the application[6].

Sven Woop, et al. also implemented a specialized hardware solution to ray tracing. Here, the authors developed an RPU (Ray Processing Unit) that resembles GPUs, but with extended functionality and optimized for ray tracing instead of rasterization. The authors describe the unit as being flexible like CPUs, but containing the parallelism of a GPU.

The design starts with a Shader Processing Unit which uses four component, single precision floating point or integer vectors for intersection tests and shading. This SPU can switch between threads whose states are maintained in hardware. Every primary ray is a separate thread, and chunks of threads are executed in SIMD mode in parallel by multiple SPUs.

Another feature, and unlike modern GPUs, is their architecture supports conditional branching, recursion, and a hardware-maintained register stack. This allows for recursively tracing rays in shaders, which certainly adds flexibility, but is probably not absolutely necessary. Another interesting feature of the architecture is the inclusion of the TPU, or Traversal Processing Unit, which works with the SPU to traverse the scene's kd-tree. This is quite interesting, because in current graphics applications, spatial divisions are contained in software; there is no notion of kd-trees, etc in GPUs. Memory access is also a nice feature in this architecture. Memory can refer to on-card DRAM or host memory, and since different rays often access the same data, these memory requests are combined. Last, this architecture is scalable to use multiple RPUs on one card, on multiple cards, or on multiple computers, much like the growing trend of multiple GPU configurations.

The authors implemented a simple prototype with a single RPU. Despite its relatively slow 66 MHz clock speed, it achieved impressive results. It can run at 1-20 frames per second (depending on scene complexity), keeping pace with multi-CPU solutions and specialized, less-flexible solutions [12].

Graphics processors have evolved quite a bit in the past few years. One of the key changes has been the addition of pixel, or fragment, shaders, which allows the programmer to directly modify a frame pixel by pixel. The shader is run on the GPU, so it can take advantage of the GPU's parallel architecture and fast floating point operations. Thus, there have been several attempts at using a pixel shader to perform ray tracing. One paper by Timothy J. Purcell et al. discusses how this could be done. The paper was written at the time when pixel shaders were first being introduced, so the authors created a simulator instead of actually implementing it.

To do ray tracing on a GPU, the authors treat the hardware like a stream processor, which means it reads data it needs as a sequential stream. Every element requires similar calculations, so the system executes a kernel on each element and places the result on an output stream. Because each element is independent, they can be processed in parallel as much as the hardware allows.

Using this stream model, the authors use several kernels that feed into one another. The first is the eye ray generator. It is the simplest kernel; it takes the camera information and creates a ray for each pixel. The second kernel is the traverser, which follows each ray and calculates which voxels are intersected. Voxels in this system are part of the accelerated spatial structure, so they store information about what objects are stored inside them. The third kernel, the intersector, then takes this information and performs ray-triangle intersection tests on all triangles in said voxels. Next, the shader calculates the color resulting from each ray, using standard shading techniques. Also, the shader is responsible for spawning secondary rays (shadow, reflection, refraction, etc).

The implementation is not of particular interest for the purpose of this project, but the way the authors mapped traditional ray tracing to the architecture of a GPU certainly is [8].

Another paper describing how ray tracing can be done on GPUs was written by Carr et al. The point of interest in this paper is how the authors do some work on the GPU and some on the CPU. This is done because both processors are good at different things. The GPU is very efficient at performing the same operations on many sets of data, while the CPU is good at recursion and handling complex data structures. Therefore, the CPU handles traversing the BSP and gathering clusters of rays and triangles to send to the GPU, and the GPU does most of the intersection tests. One of the biggest obstacles is the slow communication between CPU and GPU, so obviously the amount, and frequency, of data transferred is minimized as much as possible. To do this, the authors have the CPU gather chunks of coherent rays. If there were over a certain number of coherent rays they are sent to the GPU. Otherwise, the GPU speedup isn’t worth the time required to send the data, so the CPU performs the intersection tests itself [2].

There is another example of real-time ray tracing on a GPU on the Internet as well. Here, the author creates a ray tracer in the pixel shader of an Nvidia 8800 GPU. The scene is very basic, with 21 spheres and 1 plane. Furthermore, the ray tracer uses ray-sphere intersection instead of the standard ray-triangle intersection to reduce the number of ray intersection tests to 22 for the entire scene. However, the demo runs on an

8800GTS with 320 MB RAM at 70 frames per second at what appears to be 800x600 resolution with reflection. While simplistic, this demo certainly shows a modern GPU can, to some extent, perform high-quality ray tracing in real-time. [3]

In order to create a practical real-time ray tracer, one must consider all these factors: spatial subdivision, dynamic scenes, and parallelization. As far as the last approach (CREMA), introducing new hardware can be difficult for a number of reasons; primarily because consumers would much rather use the hardware they already have. Of course, developing and producing specialized hardware is expensive and time consuming as well. However, the basic idea of parallelizing object space instead of pixel space has much potential, and could be useful in other approaches.

Section 4: Hypothesis

Rasterization is the method used in games and most 3D interactive programs. This is made possible with Graphics Processing Units (GPUs). These specialized chips' only task is to rasterize triangles into an image using their massively parallel architecture and efficient floating-point arithmetic capabilities. Interestingly, ray tracing is slow primarily because of its reliance on floating-point operations, and it also lends itself extremely well to parallelization. Thus, it is conceivable GPUs could perform ray tracing much more efficiently than general purpose processors.

Up until a year ago, GPUs were only capable of doing one thing: drawing triangles; manufacturers gave them almost no flexibility. However, GPU manufacturer Nvidia changed this with its 8000 series GPUs. Prior to this line, in order to use a GPU fcor general computig one had to do it in a shader language such as High-Level Shader

Language (HLSL). HLSL is meant for graphics applications that allow directly modifying vertex and pixel data. So, using this for other purposes was awkward and restricting.

The architecture of the new GPUs now resembles a general parallel processor, and the drivers use it for graphics rendering. Thus, the hardware is driven much more by software. Because of this, alongside its release of the 8000 GPU line, Nvidia released a C compiler for its graphics processors called CUDA (Compute Unified Device Architecture. This means anybody can write software that uses an Nvidia 8000 series GPU as a general purpose CPU [5].

I plan to leverage this compiler and hardware to perform ray tracing in real-time. Because “real-time rendering” is not a well-defined term (5 frames per second at 256x256 resolution could be called real-time), I will use three metrics to measure the success of my system: frame rate, screen resolution, and number of objects in a scene. To be called a useful real-time renderer these three must be balanced. For example, rendering five objects in a scene at 100 frames per second does not constitute a usable graphics package. A very successful result would run at around 60 frames per second at 1024x768 resolution with a hundred thousand polygons, but I will be developing a system that maximizes all three factors as much as possible.

Section 5: Synthesis

Overview

My system has been written in C and C++. Because a CPU is more efficient than a GPU with certain things, some of my system runs on the CPU, while the core ray tracing operations runs on the GPU. Thankfully, Nvidia's CUDA SDK allows for easy cooperation with CPU and GPU. So, the GPU portion is compiled with Nvidia's C compiler and runs on an Nvidia CUDA-compatible graphics card, and the rest is written in C++ and compiled with Visual Studio 2005. Also, CUDA can cooperate with OpenGL and DirectX. Interestingly, because CUDA is still relatively new, it works with these technologies in two different, limited ways. With DirectX, CUDA can share vertex buffers, and with OpenGL it can share texture buffers. Because in my system, CUDA creates an image (or texture), and that texture needs to be rendered to the screen, I take advantage of CUDA's OpenGL interoperability.

Two desktop computers will be used for testing: one with an 8800 Ultra and another with 8800 GTS with 320 MB RAM.

I have mapped traditional ray tracing algorithm to CUDA. Also, I implemented a bounding volume hierarchy to further optimize the system. Figure 1 shows a high-level flowchart of my system:

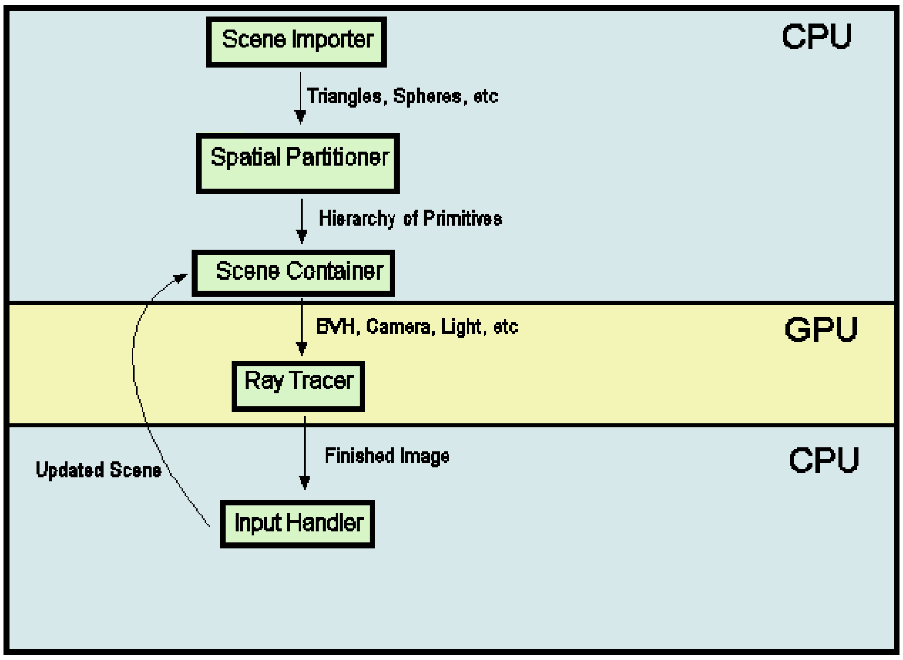


Figure 1: The pipeline of my system.

The scene geometry is initially partitioned into a Bounding Volume Hierarchy by the CPU. The CPU then uses this hierarchy to efficiently task the GPU to ray trace the scene. After the frame is complete, the CPU draws the frame using OpenGL. The CPU also checks for user input and updates the scene data (lights, camera, etc) for the next frame. While it may appear the CPU is doing most of the work, the most computationally expensive part is being done on the GPU.

Bounding Volume Hierarchy

After looking into several different techniques for spatial subdivisions, I chose to create a bounding volume hierarchy, because it seemed to be the most flexible when applied to dynamic objects. For the bounding volume I chose spheres, for reasons discussed later in this document.

Even though the hierarchy is created and maintained entirely on the host (CPU), it is sent to and used by CUDA, so I needed to consider this when designing and creating my system. Specific details on these considerations are also discussed later.

Importing geometry

A spatial subdivision is not much good without geometry, so my system allows importing models from files in the FBX format, which is a widely used format supported by Maya, 3D Studio Max, Blender, and other modeling packages. To facilitate this, I used the Autodesk FBX SDK, which provides support for reading and writing FBX files.

FBX is a very robust format, and so is the SDK. It includes support for numerous types of objects such as lights, bones, animation, NURBS, materials, cameras, etc. However, for the purpose of this project, I was only interested in polygonal geometry, so I only used features of the SDK that directly pertained to the information I wanted.

My system does not export anything to any FBX file. Therefore, when the system initializes, it creates an FBXManager and loads all relevant data. Upon completion, the FBXManager is destroyed and the SDK is not used again. An overview of the classes and structures that handle importing is described below:

ImportedModel

This class contains all relevant data from an FBX file. Since a single file can hold an entire scene, only one is currently created in my system. However, multiple files could be used with minimal amount of change to my system.

After the FBX manager has been initialized and the file opened, ImportedModel goes through the scene hierarchy. In FBX files, everything (meshes, NURBS, cameras, bones, etc) is contained in a hierarchy. Since I am only interested in the scene's geometry, this caused an issue with dealing with all the extra information. Instead of storing everything, I chose to extract mesh information and discard the rest. I also chose to discard the tree information for several reasons. The main reason was while a tree consisting entirely of meshes might be useful, in FBX files a mesh might be a child of a bone, a bone might be a child of any other type of object, etc. Thus, trying to extract a tree consisting only of meshes would be cumbersome. Also, every child has a global transformation matrix and a transformation matrix relative to its parent. Skipping parents would have invalidated a child's relative transformation matrix, so I simply always use global transformation matrices. This simplification is illustrated below:

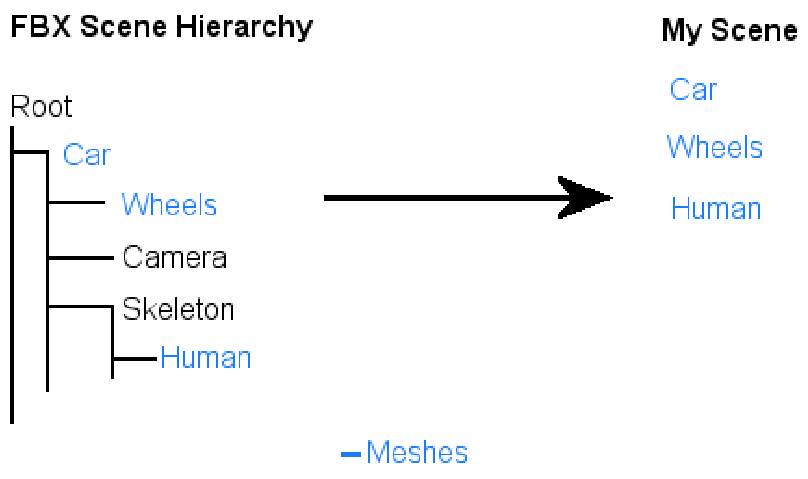


Figure 2: Example of how an FBX file is stored in my system.

ImportedModel goes through the scene hierarchy and looks for mesh objects. Whenever such an object is encountered, it creates a new Geometry object and gives it the current node. ImportedModel resumes its search at the last node's sibling.

Geometry

As mentioned above, ImportedModel gives Geometry a node that is a mesh. From there, Geometry recursively finds all offspring (children, grandchildren, etc) that are meshes and stores them as Mesh objects.

Mesh

Mesh holds a single node's vertex data. It does this by containing a series of Polygon objects. Polygons contain a number of Vertex objects, and a Vertex contains data such as x, y, z coordinates, normals, colors, etc. Mesh also contains global transformation matrices.

The diagram below shows an example of this structure.

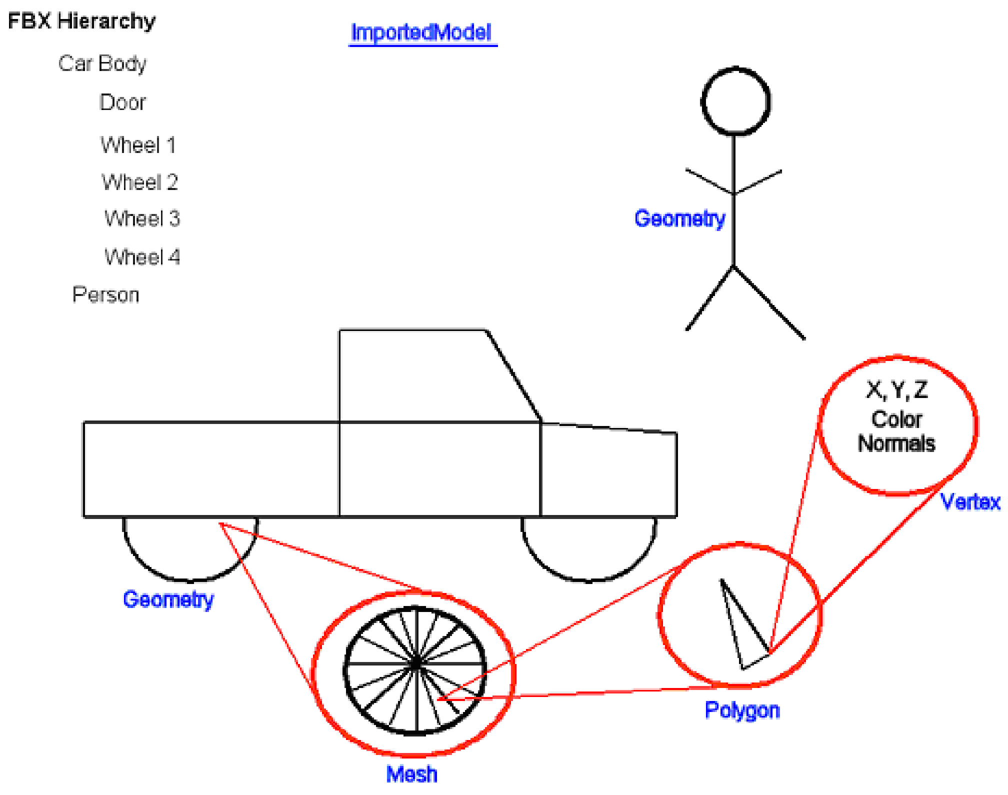


Figure 3: Examples of how a scene is stored in my class structure.

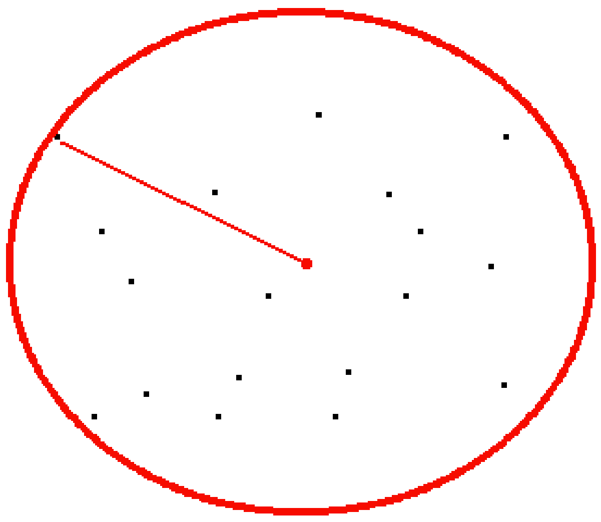
Constructing the Bounding Volume Hierarchy

Structure BoundingVolume contains data for the hierarchy. It contains its center position, radius, and transformation matrix. If this volume's children are leaf nodes it contains an array of Polygon objects; otherwise, it contains two volumes contained within it (yielding a binary tree). A BoundingVolume does not contain both child volumes and polygons.

ImportedModel constructs and contains a bounding volume hierarchy for each Mesh it contains. There are no higher-level volumes, because as these objects move, scale, or rotate, the structure of the hierarchy would need to change or become inefficient. Instead, as a single mesh is transformed, the hierarchy containing it uses the same transformation matrix. Thus, no hierarchies ever require restructuring.

So, ImportedModel loops over all Mesh objects, creating one volume for each. If the number of vertices contained in one volume exceeds a certain threshold, and if the object can be split up, the vertices are divided as evenly as possible, and a child volume is created for both groups.

Creating the bounding volume is an interesting problem. My first impulse was to find the average of all points, use that as the center, and use the distance to the farthest vertex as the radius. In this example, this approach would be acceptable:



However, this example yields an inefficient bounding volume:

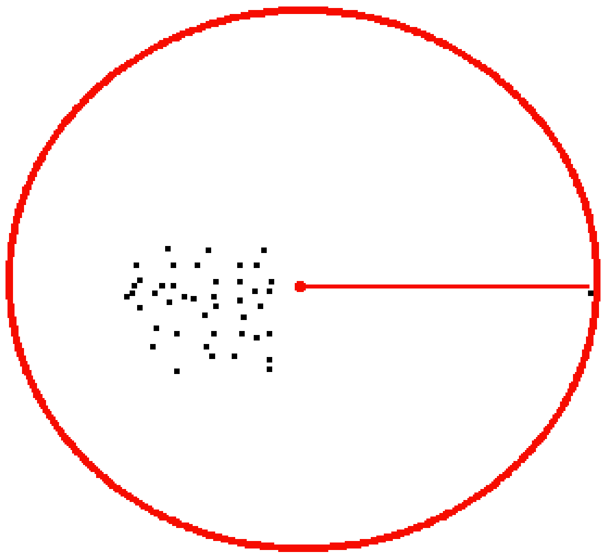


Figure 4: Inefficient bounding volume.

Therefore, I created a very simple, yet efficient way of creating a tight volume. To begin, I create a bounding box by finding the maximum x, y, and z coordinates and the minimum x, y, and z coordinates. Then, the line between these two points is the sphere's diameter, and the midpoint is the center of the sphere. With the same vertices this method yields the following bounding volume:

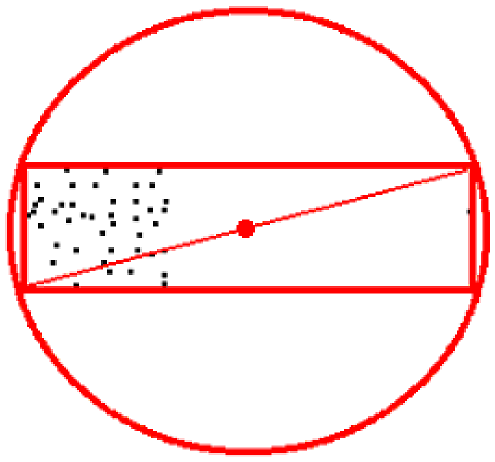


Figure 4: A more efficient bounding volume.

As noted before, transforming objects can be done by modifying each Mesh's transformation matrix. BoundingVolume contains a pointer to its Mesh's matrix, so the bounding volume hierarchy is always up to date.

One last note about bounding volumes in my system: there are two primitives that can be natively ray traced in my implementation, spheres and polygons. Incidentally, because I chose my bounding volumes to be spheres, my hierarchy contains data necessary to describe any combination of polygons and spheres. If there is a top-level volume that contains no children and no polygons, my ray tracer renders the volume itself.

Since it would have been cumbersome to export and import spheres to/from FBX files, I created my own, very simple, file format. My system reads this text file, and the file can tell the system to load an FBX file, as well as draw any number of spheres or polygons.

Considerations for CUDA

First, CUDA kernels must be written in C. This is not to say, however, the entire system must be written in C. Furthermore, the FBX SDK is written in C++. So, in my system, most of the code that runs on the host (the computer's main CPU) is C++ code. This includes ImportedScene, Geomety, and Mesh.

As previously note, the host handles the creating and updating of the hierarchy and geometry (as well as other things, such as camera control), while the GPU only handles the drawing of the scene. The GPU cannot directly access CPU memory, so the host has to explicitly send all required data to the GPU. To simplify this, BoundingVolume is a structure that contains all data required to draw a given scene. This means I am able to use C++ data types and constructs for Geometry, Mesh, etc, but BoundingVolume and its members are strictly C-type structs. The upshot of this is BoundingVolume contains pointers to matrices and children. This means I need to recursively write this data to GPU memory. To do this, I recurse down to the deepest leaf first, copy that node to the GPU memory, use that pointer for its parent, copy the parent, and so on.

Another consideration is while the GPU performs all transformation operations (matrix multiplication, translation/rotation/scale encoding and decoding) in current graphics applications, there are no mechanisms to do these operations in CUDA. Unfortunately, even though DirectX would have allowed easy access to the card's matrix operations, I had to use OpenGL for its abiltity to share textures with CUDA. Therefore, two options were available for transforming objects. First, all matrix operations could be done on the CPU, leading to losses in overall performance. The other option was to have the ray tracer kernel transform all objects as it encountered them. Obviously, this would have also greatly decreased performance. In light of this, and due to the fact that my Graduate work is going to focus on rendering performance, the objects in my scene are, for the most part, static. The mechanisms are in place to transform objects, but most of them are not used at this time. There is one exception to this, however. Models that are imported from an FBX files have transformation matrices associated with them. Simply ignoring these would yield an incorrect scene, so there is a preprocessing step that occurs. After the scene is imported, but before the bounding volume hierarchy is created, each object's transformation matrix is applied to all its vertices. These new positions simply replace the old ones.

The Ray Tracer Kernel

Overview

CUDA programs consist of kernels that are compiled specifically for the GPU. In my system, when the host is ready to draw another frame, it sends all required data to a function that spawns the GPU threads and starts up the kernel. This data includes a pointer to an OpenGL texture buffer, dimensions of the window, camera information, the scene's light position, a pointer to the bounding volume hierarchy, and some extra information about the hierarchy. The GPU cannot access any host memory and vice-versa, so all data the CPU sends is either by-value or pointers to data already in GPU memory. While in changing scenes the CPU would send the GPU the certain scene data every frame in order to reflect any changes, I ultimately made my system upload the scene geometry data once since objects in my scene do not move. I do, however send things like camera position/orientation, etc. every frame.

The spawning function then forwards this data to the kernel, while also determining how to divide the work on the GPU. This is done on every CUDA program by splitting the work into grids, and splitting grids into threads. My implementation uses ray packets (each thread computes a block of pixels), so the spawning function figures out how many grids and threads to spawn based on customizable parameters (packet size, screen size, etc). I use ray packets to decrease the number of threads that attempt to access the same memory location at one time (memory issues are discussed in further detail later in this document. Of course, this does not solve the memory access issues, but it does seem to help. The packet size is also easily configurable to allow experimenting to see how different sizes affect the performance of different scenes. On average, 4x4 packets seem to work the best.

Threads in a CUDA kernel always have access to which grid they belong to and where they are in the grid, so they can easily decide which pixel(s) they should compute. The ray tracer algorithm can then be performed on each thread.

Recursion

There is no recursion in CUDA. This posed an interesting problem, as the basic ray tracing algorithm is inherently recursive. Moreover, my bounding volume hierarchy is a tree, which is also usually traversed recursively. Creating the hierarchy wasn't a problem since it is created on the host, and thus could be done recursively, but an iterative solution was required for on CUDA. Interestingly, the solutions for both traversing the hierarchy and recursive ray tracing were very similar.

In both solutions I maintain an array of current objects and an array of “next” objects. For the hierarchy these objects are volumes, and for the ray tracing these objects are rays. So, the “current” hierarchy and ray arrays are initialized to contain the top-level volumes and the single ray spawned from the camera, respectively. Then, in every iteration, the “next” arrays are filled with child volumes whose parents were intersected and any necessary secondary rays. After each iteration the “next” array is moved to the “current” one and is emptied. For the hierarchy, iteration stops when the “current” array is empty. When ray tracing, the iterations stop when there are no more “next” rays or the number of iterations has reached a specified limit.

This iterative approach works well on CUDA, except for one other drawback. CUDA cannot dynamically allocate memory from within the kernel, so the arrays described above must be of fixed size. This means a maximum size had to be chosen and the arrays are always given this size. Of course, this means excessive memory could be allocate which will decrease performance, or not enough memory could be allocated which will cause the kernel to crash. This maximum size could be modified to allow for larger scenes, however.

Ray tracer operations

GPUs are still primarily used for rendering graphics, so they have many vector operations (dot product, cross-product, normalization, etc) implemented in hardware. CUDA provides native 2-, 3-, and 4D vector types, but the necessary vector operations are not available, which probably could have improved performance, since I wrote them all in software.

Aside from vector operations, the other main features of my kernel are the ray-sphere and ray-polygon intersection tests. The ray-sphere intersection test is the same as any other implementation, but the ray-polygon test is not as common. To begin, the polygon in question is "projected" to a 2D axis-aligned plane. This projection is not done by rotating the polygon; rather, the dominant component in the normal is simply discarded. So, if a polygon has a normal of (.1, .1, .9), the polygon is projected on the xy-plane by ignoring each vertex's z-component. Next, the polygon and intersection point are translated such that the intersection point is on the origin. For each edge on the polygon, it is tested whether it intersects a given positive axis. If the number of these intersections is even, the point lies outside the polygon; otherwise it is inside [7]. This algorithm has some advantages over others, such as it does not require any trigonometric operations and it works on both convex and non-convex polygons.

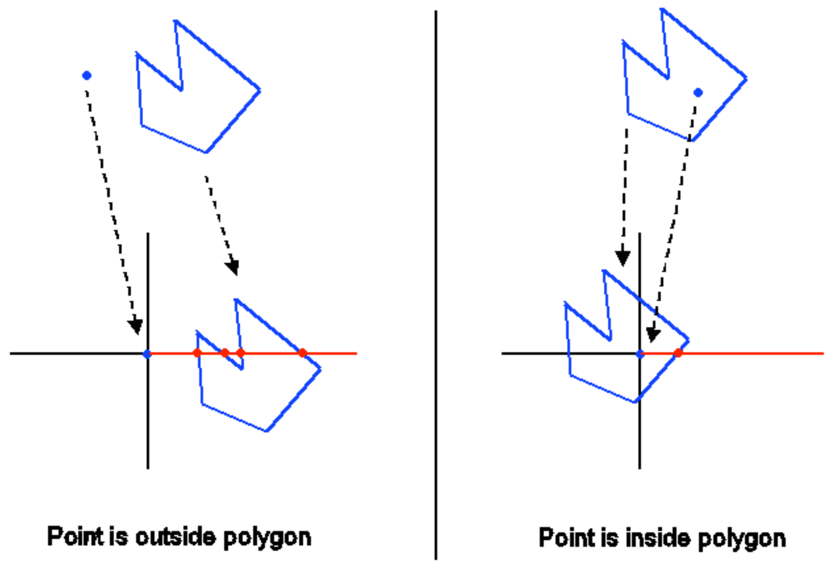


Figure 5: Process of determining if a point is inside a given polygon.

Memory Access

The 8000 GPUs have extremely powerful and parallel architectures. However, to be as powerful as they are, they have to be somewhat specialized in the types of applications they are efficient at (otherwise they could be used as a CPU!). One of the main things CUDA kernels need to have to effectively use the full capabilities of the architecture is coalesced memory. The basic idea of coalesced memory is every global memory access should be coalesced into one contiguous request, and every thread should request different memory locations. For example, if an array is being processed, thread 0 should read element 0, thread 1 should access element 1, and so on. This concept is best depicted in the CUDA Programming Guide [5]:

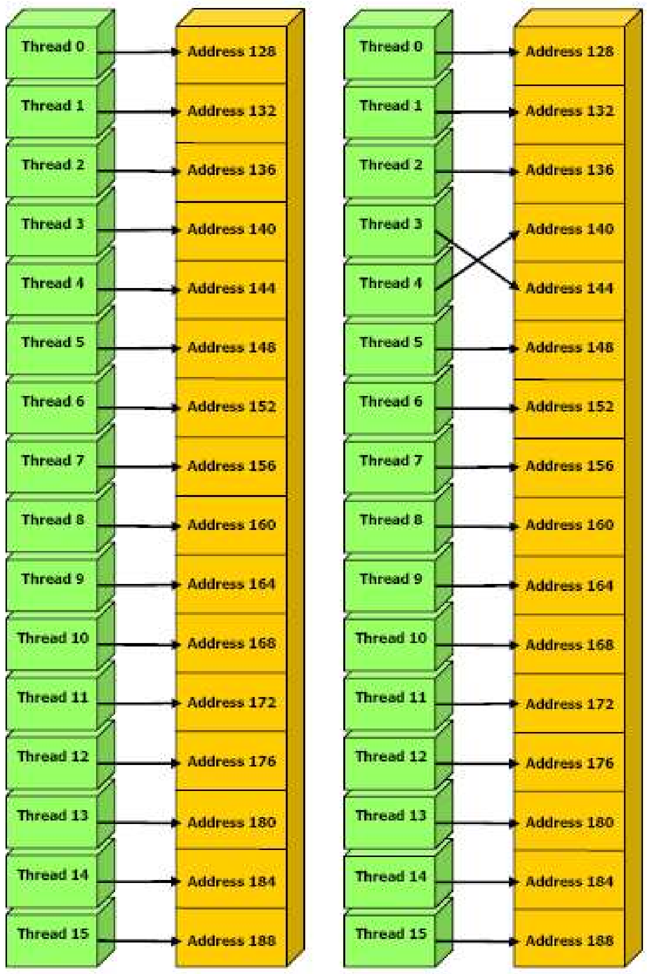


Figure 6: Left: Coalesced memory access. Right: Non-sequential non-coalesced memory access.

Unfortunately, ray tracing relies on random access memory, because each thread needs to access all scene data a number of times throughout the algorithm. This certainly degrades a ray tracer's performance when done on CUDA.

Branching

Another limiting factor in CUDA is branching. If two threads diverge, they are executed serially instead of in parallel, which obviously causes a significant decrease in performance. Again, there is much branching in ray tracing, because of different object intersections, spawning reflection/refraction rays, shadows, etc. This is a fundamental characteristic of the ray tracing algorithm, so this is another reason it performs less than ideally on CUDA.

Section 6: Results

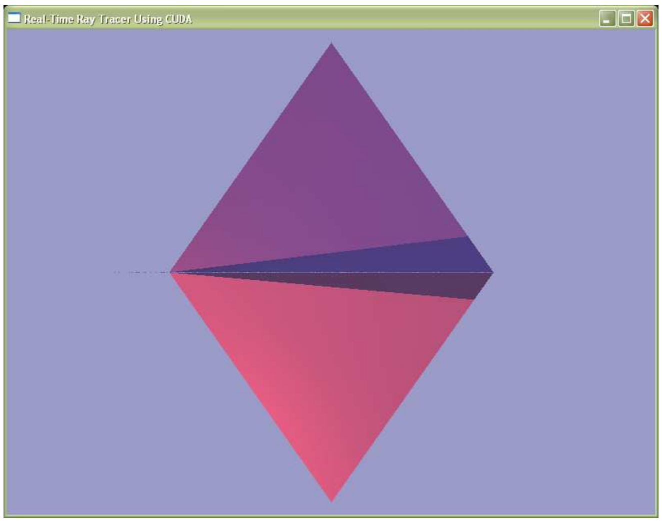
Each benchmark was run on two machines:

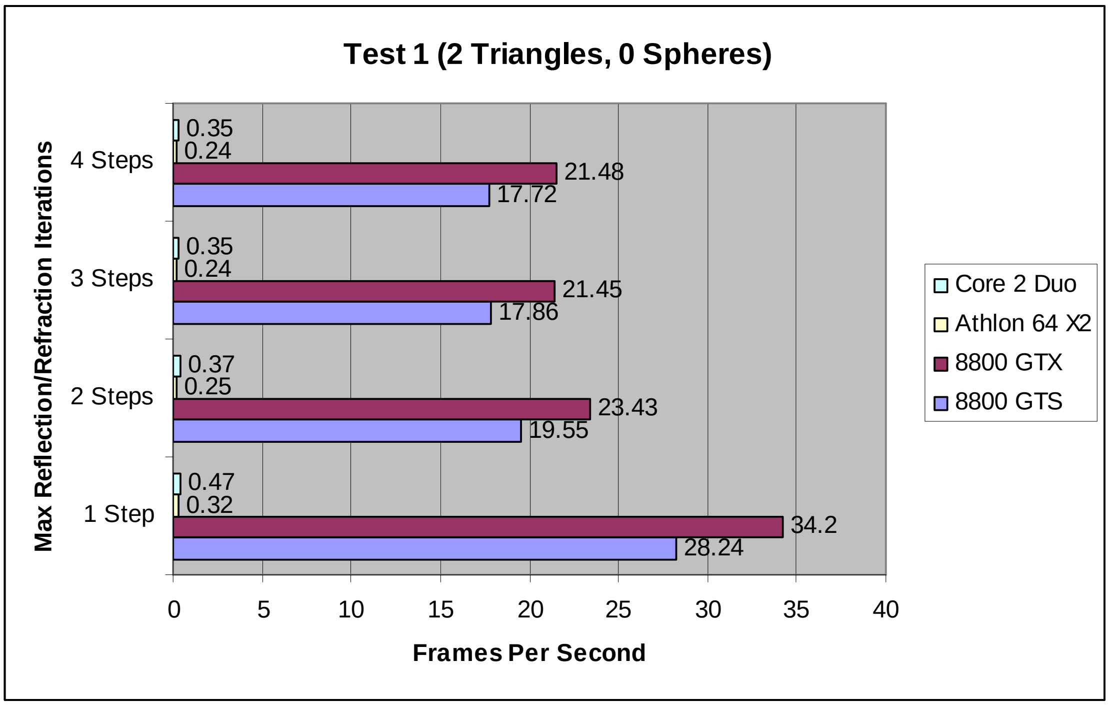
Processor	AMD ATHLON 64 X2 5200 @ 2.6 GHz	Intel Core 2 Duo E6600 @ 2.4 GHz
System RAM	4 GB	2 GB
GPU	Nvidia 8800 GTS	Nvidia 8800 GTX
Graphics RAM	320 MB	768 MB

In Addition, CUDA allows the kernel to run in "emulation" mode, which means it runs on the CPU, and the GPU is not utilized. This allows for a good measurement of the speed-up obtained from the GPU. So, tests were conducted in both emulation mode and "normal" mode. All objects in the following tests are either reflective or transparent.

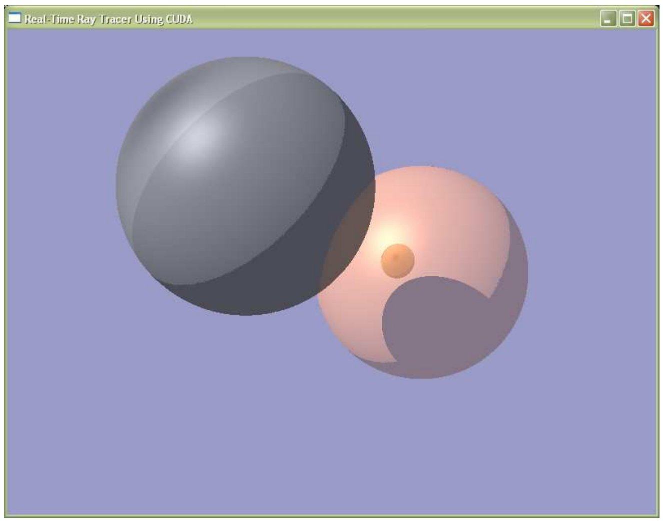
Using smaller resolutions yield linear speedups, so I have used the same resolution on all the following tests.

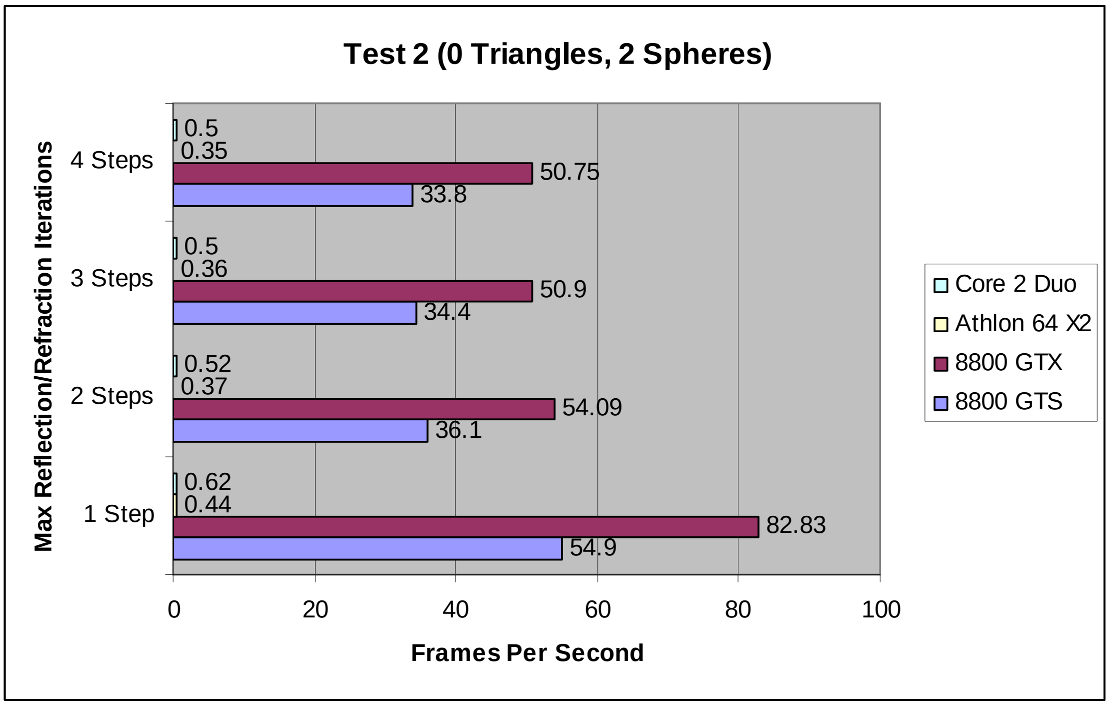
Test 1:



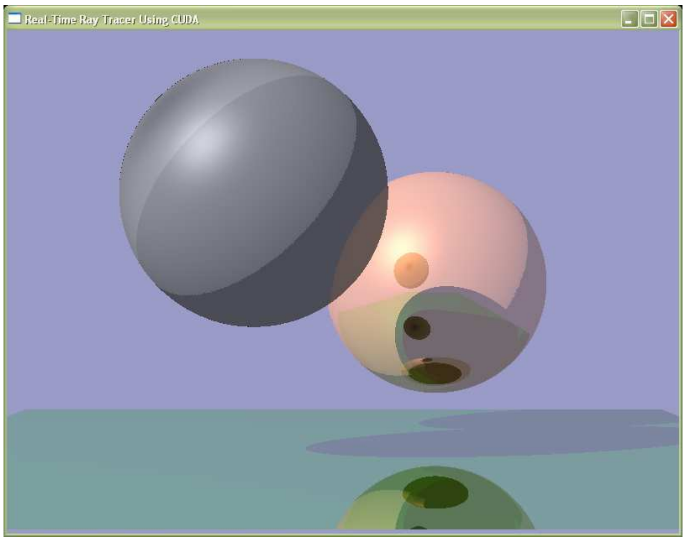


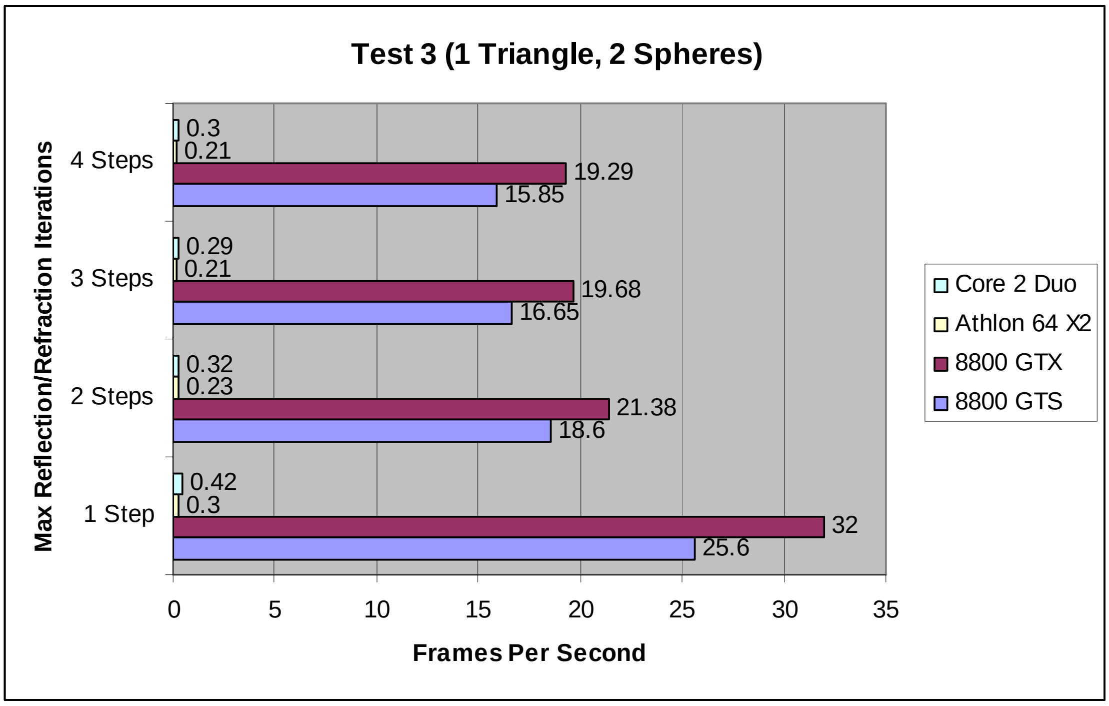
Test 2:


Real-Time Ray Tracer Using CUDA

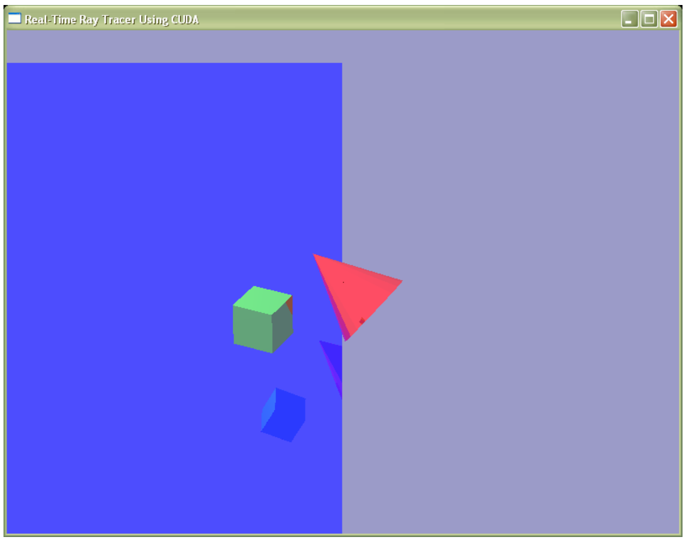

Test 2 (0 Triangles, 2 Spheres)
Max Reflection/Refraction Iterations
4 Steps
3 Steps
2 Steps
1 Step
0.5
0.35
50.75
33.8
0.5
0.36
50.9
34.4
0.52
0.37
54.09
36.1
0.62
0.44
82.83
54.9
0
20
40
60
80
100
Frames Per Second
Core 2 Duo
Athlon 64 X2
8800 GTX
8800 GTS

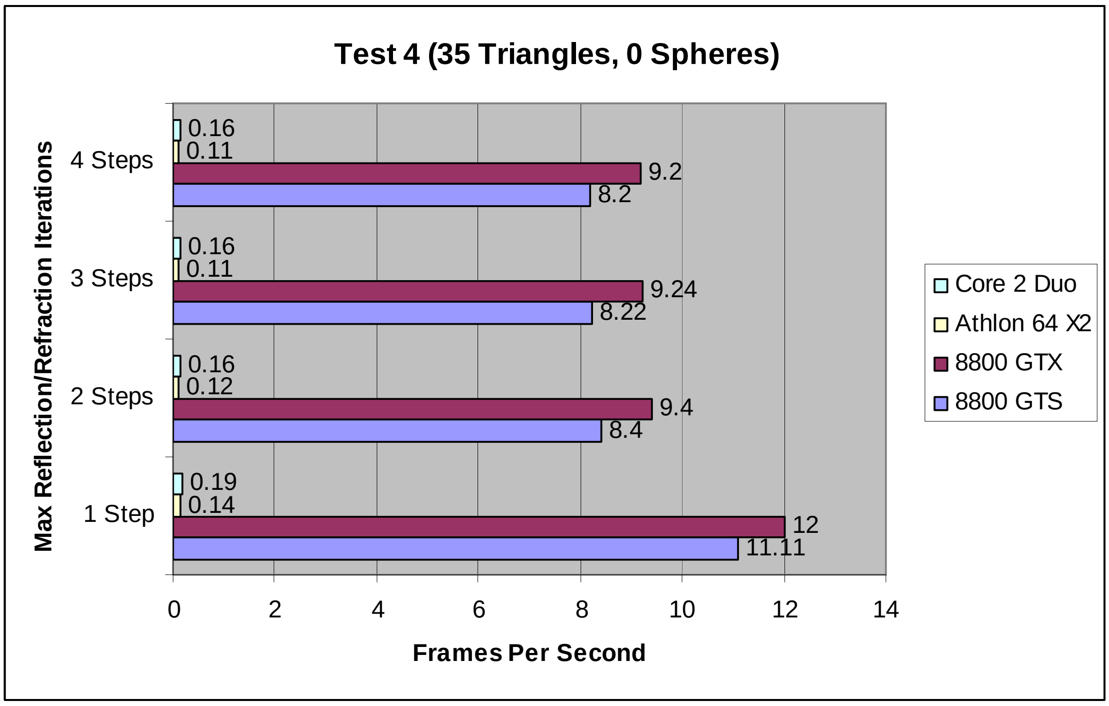
Test 3:


Real-Time Ray Tracer Using CUDA

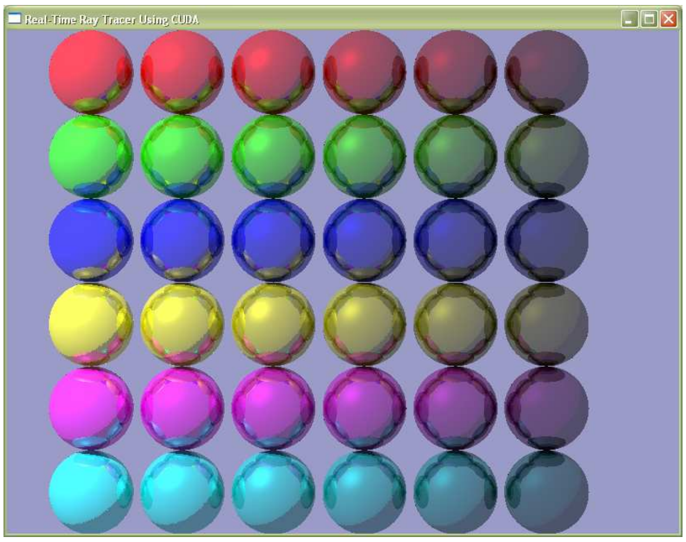

Test 3 (1 Triangle, 2 Spheres)
Max Reflection/Refraction Iterations
4 Steps
3 Steps
2 Steps
1 Step
0.3
0.21
19.29
15.85
0.29
0.21
19.68
16.65
0.32
0.23
21.38
18.6
0.42
0.3
32
25.6
0
5
10
15
20
25
30
35
Frames Per Second
Core 2 Duo
Athlon 64 X2
8800 GTX
8800 GTS

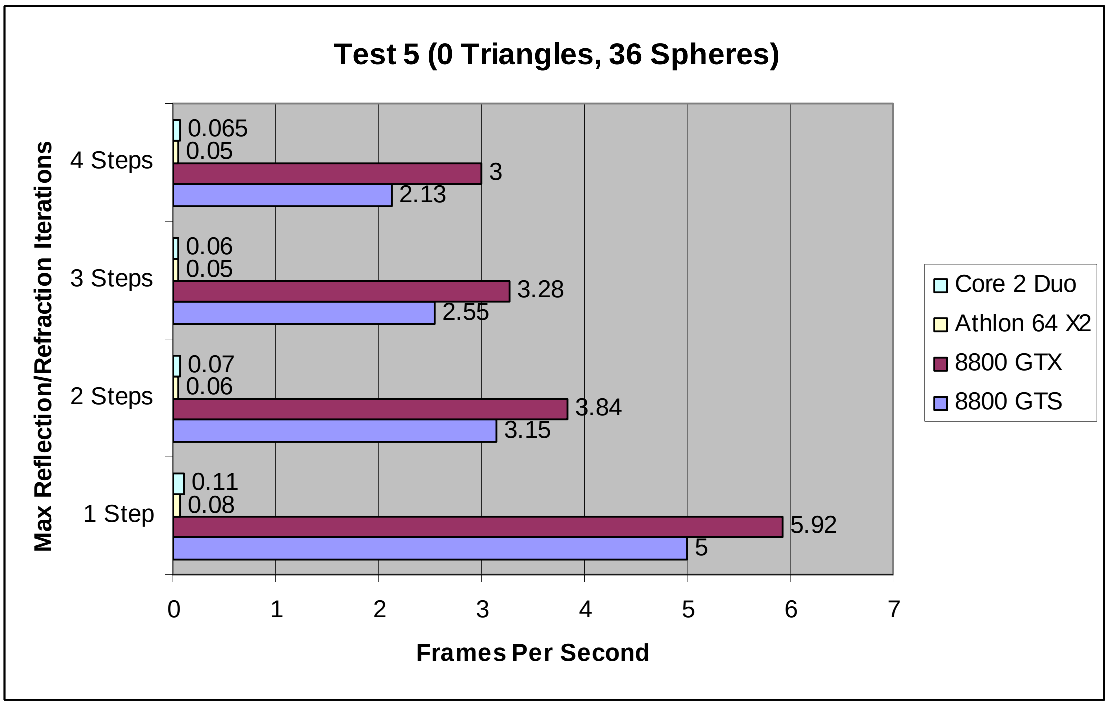
Test 4:


Real-Time Ray Tracer Using CUDA

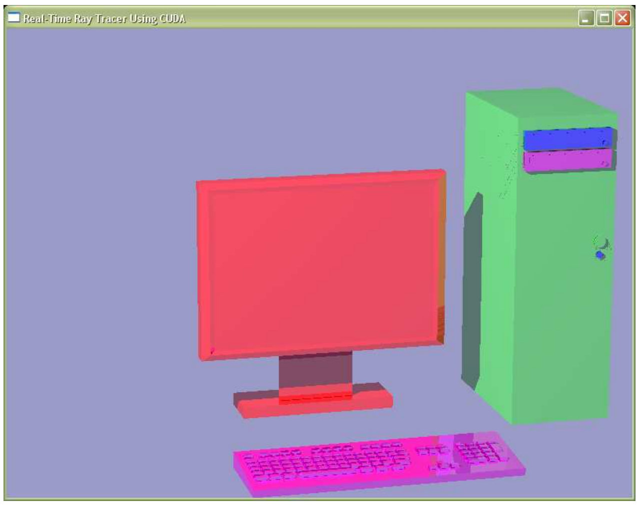

Test 4 (35 Triangles, 0 Spheres)
Max Reflection/Refraction Iterations
4 Steps
3 Steps
2 Steps
1 Step
0.16
0.11
9.2
8.2
0.16
0.11
9.24
8.22
0.16
0.12
9.4
8.4
0.19
0.14
12
11.11
0
2
4
6
8
10
12
14
Frames Per Second
Core 2 Duo
Athlon 64 X2
8800 GTX
8800 GTS

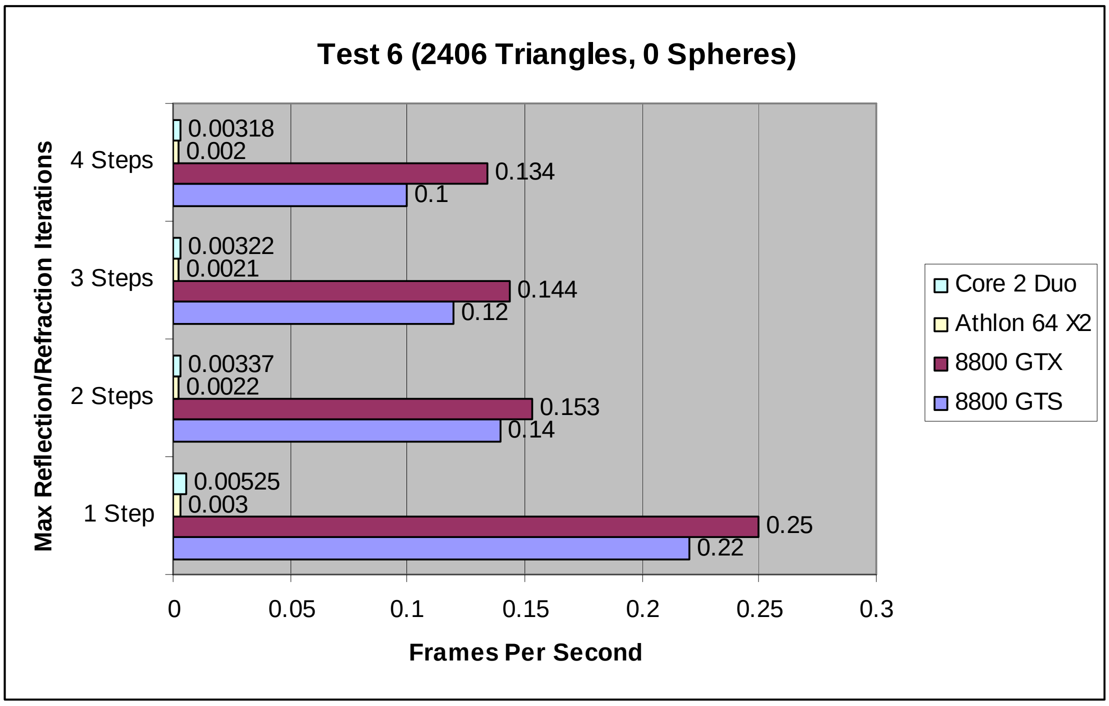
Test 5:


Real-Time Ray Tracer Using CUDA


Test 5 (0 Triangles, 36 Spheres)
Max Reflection/Refraction Iterations
4 Steps
3 Steps
2 Steps
1 Step
0.065
0.05
3
2.13
0.06
0.05
3.28
2.55
0.07
0.06
3.84
3.15
0.11
0.08
5.92
5
0
1
2
3
4
5
6
7
Frames Per Second
Core 2 Duo
Athlon 64 X2
8800 GTX
8800 GTS

Test 6:





The relative performance results are not surprising. The 8800 GTX has more memory, higher clock speeds, etc. than the GTS, so it should achieve higher frame rates. Likewise, GPUs' architectures are much more parallel than CPUs, so they should perform much better. Last, Intel's Core 2 Duo line generally performs faster than Athlon 64s.

One may be confused as to why the scene with two spheres ran so much faster than the one with two triangles when the performance of 35 polygons was better than 36 spheres. This is because my system always inserts all polygons into bounding volumes. Therefore, two triangles actually require a maximum of four intersection tests: two spheres and two polygons.

Section 6: Conclusions

Nvidia's 8800 graphics cards are extremely powerful, and CUDA allows programmers to develop software that uses these highly parallel architectures for general computing. The potential performance of floating-point arithmetic on these cards is truly impressive. However, not every problem is a good candidate for being solved with CUDA. The first and most important requirement is the problem should be massively parallelizable. Second, the solution to the problem should follow certain memory patterns. And last, branching should minimalized as much as possible.

Ray tracing requires millions of floating-point operations per second, and is also extremely parallelizable. On the other hand, it requires much random access memory and branching. Thus, ray tracing on CUDA can be done in real-time, but only on a very small scale. My system can ray trace a scene with only a handful of objects before its frame rate drops to undesirable levels. Once again, I believe the 8800 GPUs have enough computing power for ray tracing, but the memory accesses and branching are the limiting factors.

That being said, an interesting fact has been observed. On the demo programs that ship with CUDA, the GPU shows 16-300 times the performance as CPUs running the same programs in emulation mode. My system shows the GPU being about 70-100 times faster than a CPU, which is close to the gain CUDA gives to other applications, even though CUDA is not perfectly suited for ray tracing. This suggests another possible application to my system: rendering farms. Instead of using CUDA for real-time ray tracing it could be used to simply do it faster than current systems. Rendering a frame in 1/80th the time could be extremely valuable for film studios that rely on large rendering farms to produce their images.

Section 7: Future Work

There are numerous modifications or additions that could be made to this project. First, it would be interesting to investigate how other primitives such as cylinders, cubes, cones, torii, etc. perform when compared to spheres and polygons. Furthermore, NURBS can represent objects that are just as complex as those made of polygons, and since they would require fewer memory accesses, it would be extremely interesting to extend my system to accept any arbitrary NURBS or similarly-defined object. Another possible path would be to investigate how other spatial subdivisions (*kd*-trees, bsp-trees, etc) perform, as well as how other kinds of volumes for a bounding volume hierarchy affect performance. Of course, future releases of CUDA may include native matrix or vector operations, so utilizing these functions may yield performance gains. Last, seeing how multiple GPUs in SLI improve performance could be beneficial as well.

Aside from improving the performance of the system, some other features would also be quite interesting. Because the hardware is designed for graphics after all, and because CUDA cooperates with OpenGl and DirectX, using the pixel/fragment shaders for post-processing is a possibility as well. Effects like tone reproduction could be implemented in a shader after CUDA finishes tracing the scene.

Appendix A: Installing and configuring necessary software for project on Windows XP and Visual Studio 2005:

Nvidia CUDA:

Download and install three files from http://www.nvidia.com/object/cuda_get.html#windows:

- 169.21_forceware_winxp_32bit_english_whql.exe
- NVIDIA_CUDA_Toolkit_1.1_x86.exe
- NVIDIA_CUDA_SDK_1.1_x86.exe

Autodesk FBX SDK:

Download fbx200611_1_fbxsdk_win_enu.exe from http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=6839916

Running my Project:

Note: These steps assume the above installations were performed with the default options. If not, the paths for the FBX SDK and CUDA may require changing in Visual Studio.

- Copy my (unzipped) project directory to C:\Program Files\NVIDIA Corporation\NVIDIA CUDA SDK\projects\raytracer, such that raytracer.sln is immediately contained in this directory.
- Open ratracer.sln in MS Visual Studio 2005.
- Build solution.
- If everything compiles and links, run project. Otherwise, refer to “troubleshooting” below.

Controls:

- Arrow keys move the camera in the X and Z direction
- I, K Adjusts camera pitch
- O, P Adjusts camera yaw
- W, S, A, D move the light source in the X and Z direction
- F prints the current frame rate to the console
- 1-9 displays respective levels of bounding volume hierarchy. 0 displays the bottom level.

Troubleshooting

- `>LINK : fatal error LNK1181: cannot open input file 'fbxsdk_mt2005d.lib'`
    - You may need to include the FBX “lib” directory in Visual Studio.
        - In VS, go to Tools -->options
        - Under Projects and Settings, click VC++ directories
        - Select Library Files from the “show directories for” drop-down

        - Add C:\Program Files\Autodesk\FBX\FbxSdk\2006.11.1\lib to the list.

- `>C:\Program Files\Autodesk\FBX\FbxSdk\2006.11.1\include\kfcurve/kfcurvenode.h(1056) : fatal error C1083: Cannot open include file: 'kfcurve/kfcurvenodeinhouse.h': No such file or directory`
    - Double-click the error. This should open the .h file with the error. Comment out line 1056.
    - Rebuilding will give 329 warnings, but the program will run fine. Unfortunately, no other reliable fix has been found that does not yield these warnings.

For more accurate performance statistics, disable Vertical Sync:

- Open Nvidia Control Panel
- Click “Manage 3D Settings”
- In the “Global Settings” tab, select “Force off” for the “Vertical Sync” feature

References

[1]*Binary Space Partitioning.* 8 October 2007. Retrieved 20 October 2007 from site: http://en.wikipedia.org/wiki/BSP_tree.

[2] Carr, N. A., Hall, J. D., and Hart, J. C. 2002. The ray engine. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (Saarbrucken, Germany, September 01 - 02, 2002). SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware. Eurographics Association, Aire-la-Ville, Switzerland, 37-46.

[3] *Hardware RayTracing Demo for IrrSpintz – nVidia 8800*. 26 May 2007 Retrieved 21 October 2007, from site: http://sio2.g0dsoft.com/modules/wmpdownloads/ .

[4]*Kd-tree.* 17 October 2007. Retrieved 20 October 2007 from site: http://en.wikipedia.org/wiki/Kd-tree.

[5] Nvidia. NVIDIA CUDA Complete Unified Device. Version 1.1. Programming Guide. 29 November 2007.

[6]Ochsenfahrt, Ulf; Salomon, Ralf, "CREMA: A Parallel Hardware Raytracing Machine," *Circuits and Systems, 2007. ISCAS 2007. IEEE International Symposium on* , vol., no., pp.769-772, 27-30 May 2007.

[7] Owen, Scott. *Ray – Polygon Intersection.* 2 June 1999. Retrieved 14 April 2008 from site: http://www.siggraph.org/education/materials/HyperGraph/raytrace/raypolygon_intersection.htm.

[8] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. 2002. Ray tracing on programmable graphics hardware. In *Proceedings of the 29th Annual Conference on*

Computer Graphics and interactive Techniques (San Antonio, Texas, July 23 - 26, 2002). SIGGRAPH '02. ACM, New York, NY, 703-712. DOI= http://doi.acm.org/10.1145/566570.566640.

[9]*Scene Graph.* 8 September 2007. Retrieved 20 October 2007 from site: http://en.wikipedia.org/wiki/Bounding_volume_hierarchies.

[10]Wald, I. 2005. Handling dynamic scenes. In *ACM SIGGRAPH 2005 Courses* (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM Press, New York, NY, 14. DOI= http://doi.acm.org/10.1145/1198555.1198753.

[11]Wald, I., Boulos, S., and Shirley, P. 2007. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.* 26, 1 (Jan. 2007), 6. DOI= http://doi.acm.org/10.1145/1189762.1206075.

[12] Woop, Sven, Schmittler, Jorg, and Slusallek, Philipp. “RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing,” Proceedings of ACM SIGGRAPH 2005. July 2005. DOI=http://www.saarcor.de/.