Ray Tracing on Programmable Graphics Hardware

Timothy J. Purcell   Ian Buck   William R. Mark [*]   Pat Hanrahan

Stanford University [†]

Abstract

Recently a breakthrough has occurred in graphics hardware: fixed function pipelines have been replaced with programmable vertex and fragment processors. In the near future, the graphics pipeline is likely to evolve into a general programmable stream processor capable of more than simply feed-forward triangle rendering.

In this paper, we evaluate these trends in programmability of the graphics pipeline and explain how ray tracing can be mapped to graphics hardware. Using our simulator, we analyze the performance of a ray casting implementation on next generation programmable graphics hardware. In addition, we compare the performance difference between non-branching programmable hardware using a multipass implementation and an architecture that supports branching. We also show how this approach is applicable to other ray tracing algorithms such as Whitted ray tracing, path tracing, and hybrid rendering algorithms. Finally, we demonstrate that ray tracing on graphics hardware could prove to be faster than CPU based implementations as well as competitive with traditional hardware accelerated feed-forward triangle rendering.

CR Categories: I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

Keywords: Programmable Graphics Hardware, Ray Tracing

1 Introduction

Real-time ray tracing has been a goal of the computer-graphics community for many years. Recently VLSI technology has reached the point where the raw computational capability of a single chip is sufficient for real-time ray tracing. Real-time ray tracing has been demonstrated on small scenes on a single general-purpose CPU with SIMD floating point extensions [Wald et al. 2001b], and for larger scenes on a shared memory multiprocessor [Parker et al. 1998; Parker et al. 1999] and a cluster [Wald et al. 2001b; Wald et al. 2001a]. Various efforts are under way to develop chips specialized for ray tracing, and ray tracing chips that accelerate off-line rendering are commercially available [Hall 2001]. Given that real-time ray tracing is possible in the near future, it is worthwhile to study implementations on different architectures with the goal of providing maximum performance at the lowest cost.

[*]Currently at NVIDIA Corporation

[†]{tpurcell, ianbuck, billmark, hanrahan}@graphics.stanford.edu

In this paper, we describe an alternative approach to real-time ray tracing that has the potential to out perform CPU-based algorithms without requiring fundamentally new hardware: using commodity programmable graphics hardware to implement ray tracing. Graphics hardware has recently evolved from a fixed-function graphics pipeline optimized for rendering texture-mapped triangles to a graphics pipeline with programmable vertex and fragment stages. In the near-term (next year or two) the graphics processor (GPU) fragment program stage will likely be generalized to include floating point computation and a complete, orthogonal instruction set. These capabilities are being demanded by programmers using the current hardware. As we will show, these capabilities are also sufficient for us to write a complete ray tracer for this hardware. As the programmable stages become more general, the hardware can be considered to be a general-purpose stream processor. The stream processing model supports a variety of highly-parallelizable algorithms, including ray tracing.

In recent years, the performance of graphics hardware has increased more rapidly than that of CPUs. CPU designs are optimized for high performance on sequential code, and it is becoming increasingly difficult to use additional transistors to improve performance on this code. In contrast, programmable graphics hardware is optimized for highly-parallel vertex and fragment shading code [Lindholm et al. 2001]. As a result, GPUs can use additional transistors much more effectively than CPUs, and thus sustain a greater rate of performance improvement as semiconductor fabrication technology advances.

The convergence of these three separate trends – sufficient raw performance for single-chip real-time ray tracing; increasing GPU programmability; and faster performance improvements on GPUs than CPUs – make GPUs an attractive platform for real-time ray tracing. GPU-based ray tracing also allows for hybrid rendering algorithms; e.g. an algorithm that starts with a Z-buffered rendering pass for visibility, and then uses ray tracing for secondary shadow rays. Blurring the line between traditional triangle rendering and ray tracing allows for a natural evolution toward increased realism.

In this paper, we show how to efficiently implement ray tracing on GPUs. The paper contains three main contributions:

- We show how ray tracing can be mapped to a stream processing model of parallel computation. As part of this mapping, we describe an efficient algorithm for mapping the innermost ray-triangle intersection loop to multiple rendering passes. We then show how the basic ray caster can be extended to include shadows, reflections, and path tracing.

- We analyze the streaming GPU-based ray caster's performance and show that it is competitive with current CPU-based ray casting. We also show initial results for a system including secondary rays. We believe that in the near future, GPU-based ray tracing will be much faster than CPU-based ray tracing.

- To guide future GPU implementations, we analyze the compute and memory bandwidth requirements of ray casting on GPUs. We study two basic architectures: one architecture without branching that requires multiple passes, and another with branching that requires only a single pass. We show that the single pass version requires significantly less bandwidth, and is compute-limited. We also analyze the performance of the texture cache when used for ray casting and show that it is very effective at reducing bandwidth.

2 Programmable Graphics Hardware

2.1 The Current Programmable Graphics Pipeline

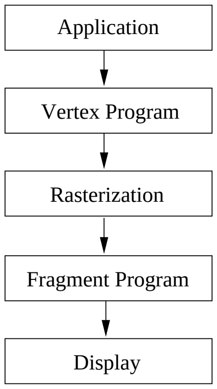

Figure 1: The programmable graphics pipeline.

A diagram of a modern graphics pipeline is shown in figure 1. Today's graphics chips, such as the NVIDIA GeForce3 [NVIDIA 2001] and the ATI Radeon 8500 [ATI 2001] replace the fixed-function vertex and fragment (including texture) stages with programmable stages. These programmable vertex and fragment engines execute user-defined programs and allow fine control over shading and texturing calculations. An NVIDIA vertex program consists of up to 128 4-way SIMD floating point instructions [Lindholm et al. 2001]. This vertex program is run on each incoming vertex and the computed results are passed on to the rasterization stage. The fragment stage is also programmable, either through NVIDIA register combiners [Spitzer 2001] or DirectX 8 pixel shaders [Microsoft 2001]. Pixel shaders, like vertex programs, provide a 4-way SIMD instruction set augmented with instructions for texturing, but unlike vertex programs operate on fixed-point values. In this paper, we will be primarily interested in the programmable fragment pipeline; it is designed to operate at the system fill rate (approximately 1 billion fragments per second).

Programmable shading is a recent innovation and the current hardware has many limitations:

- Vertex and fragment programs have simple, incomplete instruction sets. The fragment program instruction set is much simpler than the vertex instruction set.
- Fragment program data types are mostly fixed-point. The input textures and output framebuffer colors are typically 8-bits per color component. Intermediate values in registers have only slightly more precision.
- There are many resource limitations. Programs have a limited number of instructions and a small number of registers. Each stage has a limited number of inputs and outputs (e.g. the number of outputs from the vertex stage is constrained by the number of vertex interpolants).
- The number of active textures and the number of dependent textures is limited. Current hardware permits certain instructions for computing texture addresses only at certain points within the program. For example, a DirectX 8 PS 1.4 pixel shader has two stages: a first texture addressing stage consisting of four texture fetch instructions followed by eight color blending instructions, and then a color computation stage consisting of additional texture fetches followed by color combining arithmetic. This programming model permits a single level of dependent texturing.
- Only a single color value may be written to the framebuffer in each pass.
- Programs cannot loop and there are no conditional branching instructions.

2.2 Proposed Near-term Programmable Graphics Pipeline

The limitations of current hardware make it difficult to implement ray tracing in a fragment program. Fortunately, due to the interest in programmable shading for mainstream game applications, programmable pipelines are rapidly evolving and many hardware and software vendors are circulating proposals for future hardware. In fact, many of the current limitations are merely a result of the fact that they represent the very first generation of programmable hardware. In this paper, we show how to implement a ray tracer on an extended hardware model that we think approximates hardware available in the next year or two. Our model is based loosely on proposals by Microsoft for DirectX 9.0 [Marshall 2001] and by 3DLabs for OpenGL 2.0 [3DLabs 2001].

Our target baseline architecture has the following features:

- A programmable fragment stage with floating point instructions and registers. We also assume floating point texture and framebuffer formats.
- Enhanced fragment program assembly instructions. We include instructions which are now only available at the vertex level. Furthermore, we allow longer programs; long enough so that our basic ray tracing components may be downloaded as a single program (our longest program is on the order of 50 instructions).
- Texture lookups are allowed anywhere within a fragment program. There are no limits on the number of texture fetches or levels of texture dependencies within a program.
- Multiple outputs. We allow 1 or 2 floating point RGBA (4-vectors) to be written to the framebuffer by a fragment program. We also assume the fragment program can render directly to a texture or the stencil buffer.

We consider these enhancements a natural evolution of current graphics hardware. As already mentioned, all these features are actively under consideration by various vendors.

At the heart of any efficient ray tracing implementation is the ability to traverse an acceleration structure and test for an intersection of a ray against a list of triangles. Both these abilities require a looping construct. Note that the above architecture does not include data-dependent conditional branching in its instruction set. Despite this limitation, programs with loops and conditionals can be mapped to this baseline architecture using the multipass rendering technique presented by Peercy et al. [2000]. To implement a conditional using their technique, the conditional predicate is first evaluated using a sequence of rendering passes, and then a stencil bit is set to true or false depending on the result. The body of the conditional is then evaluated using additional rendering passes, but values are only written to the framebuffer if the corresponding fragment's stencil bit is true.

Although their algorithm was developed for a fixed-function graphics pipeline, it can be extended and used with a programmable pipeline. We assume the addition of two hardware features to make the Peercy et al. algorithm more efficient: direct setting of stencil bits and an early fragment kill similar to Z occlusion culling [Kirk 2001]. In the standard OpenGL pipeline, stencil bits may be set by testing the alpha value. The alpha value is computed by the fragment program and then written to the framebuffer. Setting the stencil bit from the computed alpha value requires an additional pass. Since fragment programs in our baseline architecture can modify the stencil values directly, we can eliminate this extra pass. Another important rendering optimization is an early fragment kill. With an early fragment kill, the depth or stencil test is executed before the fragment program stage and the fragment program is executed only if the fragment passes the stencil test. If the stencil bit is false, no instructions are executed and no texture or framebuffer bandwidth is used (except to read the 8-bit stencil value). Using the combination of these two techniques, multipass rendering using large fragment programs under the control of the stencil buffer is quite efficient.

As we will see, ray tracing involves significant looping. Although each rendering pass is efficient, extra passes still have a cost; each pass consumes extra bandwidth by reading and writing intermediate values to texture (each pass also requires bandwidth to read stencil values). Thus, fewer resources would be used if these inner loops over voxels and triangles were coalesced into a single pass. The obvious way to do this would be to add branching to the fragment processing hardware. However, adding support for branching increases the complexity of the GPU hardware. Non-branching GPUs may use a single instruction stream to feed several fragment pipelines simultaneously (SIMD computation). GPUs that support branching require a separate instruction stream for each processing unit (MIMD computation). Therefore, graphics architects would like to avoid branching if possible. As a concrete example of this trade off, we evaluate the efficiency of ray casting on two architectures, one with and one without branching:

- **Multipass Architecture**. Supports arbitrary texture reads, floating-point texture and framebuffer formats, general floating point instructions, and two floating point 4-vector outputs. Branching is implemented via multipass rendering.

- **Branching Architecture**. Multipass architecture enhanced to include support for conditional branching instructions for loops and control flow.

2.3 The Streaming Graphics Processor Abstraction

As the graphics processor evolves to include a complete instruction set and larger data types, it appears more and more like a general-purpose processor. However, the challenge is to introduce programmability without compromising performance, for otherwise the GPU would become more like the CPU and lose its cost-performance advantages. In order to guide the mapping of new applications to graphics architectures, we propose that we view next-generation graphics hardware as a *streaming processor*. Stream processing is not a new idea. Media processors transform streams of digital information as in MPEG video decode. The IMAGINE processor is an example of a general-purpose streaming processor [Khailany et al. 2000].

Streaming computing differs from traditional computing in that the system reads the data required for a computation as a sequential *stream* of elements. Each element of a stream is a record of data requiring a similar computation. The system executes a program or *kernel* on each element of the input stream placing the result on an output stream. In this sense, a programmable graphics processor executes a vertex program on a stream of vertices, and a fragment program on a stream of fragments. Since, for the most part we are ignoring vertex programs and rasterization, we are treating the graphics chip as basically a streaming fragment processor.

The streaming model of computation leads to efficient implementations for three reasons. First, since each stream element's computation is independent from any other, designers can add additional pipelines that process elements of the stream in parallel. Second, kernels achieve high arithmetic intensity. That is, they perform a lot of computation per small fixed-size record. As a result the computation to memory bandwidth ratio is high. Third, streaming hardware can hide the memory latency of texture fetches by using prefetching [Torborg and Kajiya 1996; Anderson et al. 1997; Igehy et al. 1998]. When the hardware fetches a texture for a fragment, the fragment registers are placed in a FIFO and the fragment processor starts processing another fragment. Only after the texture is fetched does the processor return to that fragment. This method of hiding latency is similar to multithreading [Alverson et al. 1990] and works because of the abundant parallelism in streams. In summary, the streaming model allows graphics hardware to exploit parallelism, to utilize bandwidth efficiently, and to hide memory latency. As a result, graphics hardware makes efficient use of VLSI resources.

The challenge is then to map ray tracing onto a streaming model of computation. This is done by breaking the ray tracer into kernels. These kernels are chained together by streams of data, originating from data stored in textures and the framebuffer.

3 Streaming Ray Tracing

In this section, we show how to reformulate ray tracing as a streaming computation. A flow diagram for a streaming ray tracer is found in figure 2.

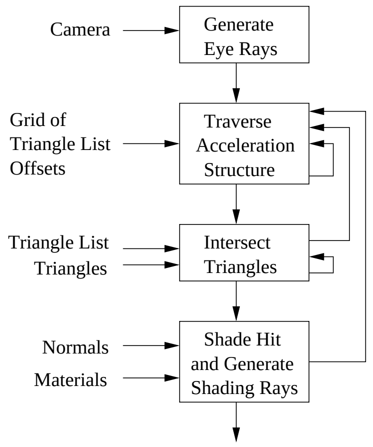
Camera
Generate Eye Rays
Grid of Triangle List Offsets
Traverse Acceleration Structure
Triangle List
Triangles
Intersect Triangles
Normals
Materials
Shade Hit and Generate Shading Rays

Figure 2: A streaming ray tracer.

In this paper, we assume that all scene geometry is represented as triangles stored in an acceleration data structure before rendering begins. In a typical scenario, an application would specify the scene geometry using a display list, and the graphics library would place the display list geometry into the acceleration data structure. We will not consider the cost of building this data structure. Since this may be an expensive operation, this assumption implies that the algorithm described in this paper may not be efficient for dynamic scenes.

The second design decision was to use a uniform grid to accelerate ray tracing. There are many possible acceleration data structures to choose from: bounding volume hierarchies, bsp trees, k-d trees, octrees, uniform grids, adaptive grids, hierarchical grids, etc. We chose uniform grids for two reasons. First, many experiments have been performed using different acceleration data structures on different scenes (for an excellent recent study see Havran et al. [2000]). From these studies no single acceleration data structure appears to be most efficient; all appear to be within a factor of two of each other. Second, uniform grids are particularly simple for hardware implementations. Accesses to grid data structures require constant time; hierarchical data structures, in contrast, require variable time per access and involve pointer chasing. Code for grid traversal is also very simple and can be highly optimized in hardware. In our system, a grid is represented as a 3D texture map, a memory organization currently supported by graphics hardware. We will discuss further the pros and cons of the grid in section 5.

We have split the streaming ray tracer into four kernels: eye ray generation, grid traversal, ray-triangle intersection, and shading. The eye ray generator kernel produces a stream of viewing rays. Each viewing ray is a single ray corresponding to a pixel in the image. The traversal kernel reads the stream of rays produced by the eye ray generator. The traversal kernel steps rays through the grid until the ray encounters a voxel containing triangles. The ray and voxel address are output and passed to the intersection kernel. The intersection kernel is responsible for testing ray intersections with all the triangles contained in the voxel. The intersector has two types of output. If ray-triangle intersection (hit) occurs in that voxel, the ray and the triangle that is hit is output for shading. If no hit occurs, the ray is passed back to the traversal kernel and the search for voxels containing triangles continues. The shading kernel computes a color. If a ray terminates at this hit, then the color is written to the accumulated image. Additionally, the shading kernel may generate shadow or secondary rays; in this case, these new rays are passed back to the traversal stage.

We implement ray tracing kernels as fragment programs. We execute these programs by rendering a screen-sized rectangle. Constant inputs are placed within the kernel code. Stream inputs are fetched from screen-aligned textures. The results of a kernel are then written back into textures. The stencil buffer controls which fragments in the screen-sized rectangle and screen-aligned textures are active. The 8-bit stencil value associated with each ray contains the ray's state. A ray's state can be *traversing*, *intersecting*, *shading*, or *done*. Specifying the correct stencil test with a rendering pass, we can allow the kernel to be run on only those rays which are in a particular state.

The following sections detail the implementation of each ray tracing kernel and the memory layout for the scene. We then describe several variations including ray casting, Whitted ray tracing [Whitted 1980], path tracing, and shadow casting.

3.1 Ray Tracing Kernels

3.1.1 Eye Ray Generator

The eye ray generator is the simplest kernel of the ray tracer. Given camera parameters, including viewpoint and a view direction, it computes an eye ray for each screen pixel. The fragment program is invoked for each pixel on the screen, generating an eye ray for each. The eye ray generator also tests the ray against the scene bounding box. Rays that intersect the scene bounding box are processed further, while those that miss are terminated.

3.1.2 Traverser

The traversal stage searches for voxels containing triangles. The first part of the traversal stage sets up the traversal calculation. The second part steps along the ray enumerating those voxels pierced by the ray. Traversal is equivalent to 3D line drawing and has a per-ray setup cost and a per-voxel rasterization cost.

We use a 3D-DDA algorithm [Fujimoto et al. 1986] for this traversal. After each step, the kernel queries the grid data structure which is stored as a 3D texture. If the grid contains a null pointer, then that voxel is empty. If the pointer is not null, the voxel contains triangles. In this case, a ray-voxel pair is output and the ray is marked so that it is tested for intersection with the triangles in that voxel.

Implementing the traverser loop on the multipass architecture requires multiple passes. The once per ray setup is done as two passes and each step through a voxel requires an additional pass. At the end of each pass, the fragment program must store all the stepping parameters used within the loop to textures, which then must be read for the next pass. We will discuss the multipass implementation further after we discuss the intersection stage.

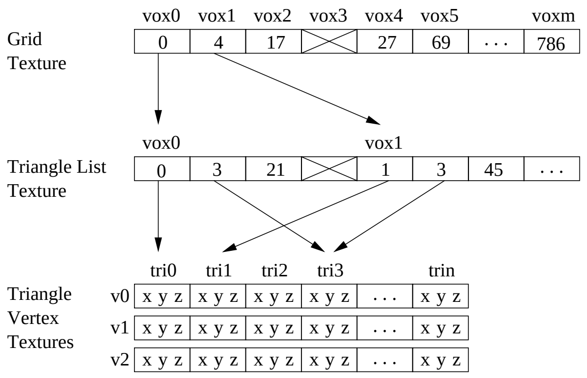

Figure 4: The grid and triangle data structures stored in texture memory. Each grid cell contains a pointer to a list of triangles. If this pointer is null, then no triangles are stored in that voxel. Grids are stored as 3D textures. Triangle lists are stored in another texture. Voxels containing triangles point to the beginning of a triangle list in the triangle list texture. The triangle list consists of a set of pointers to vertex data. The end of the triangle list is indicated by a null pointer. Finally, vertex positions are stored in textures.

3.1.3 Intersector

The triangle intersection stage takes a stream of ray-voxel pairs and outputs ray-triangle hits. It does this by performing ray-triangle intersection tests with all the triangles within a voxel. If a hit occurs, a ray-triangle pair is passed to the shading stage. The code for computing a single ray-triangle intersection is shown in figure 5. The code is similar to that used by Carr et al. [2002] for their DirectX 8 PS 1.4 ray-triangle intersector. We discuss their system further in section 5.

Because triangles can overlap multiple grid cells, it is possible for an intersection point to lie outside the current voxel. The intersection kernel checks for this case and treats it as a miss. Note that rejecting intersections in this way may cause a ray to be tested against the same triangle multiple times (in different voxels). It is possible to use a mailbox algorithm to prevent these extra intersection calculations [Amanatides and Woo 1987], but mailboxing is difficult to implement when multiple rays are traced in parallel.

The layout of the grid and triangles in texture memory is shown in figure 4. As mentioned above, each voxel contains an offset into a triangle-list texture. The triangle-list texture contains a delimited list of offsets into triangle-vertex textures. Note that the triangle-list texture and the triangle-vertex textures are 1D textures. In fact, these textures are being used as a random-access read-only memory. We represent integer offsets as 1-component floating point textures and vertex positions in three floating point RGB textures. Thus, theoretically, four billion triangles could be addressed in texture memory with 32-bit integer addressing. However, much less texture memory is actually available on current graphics cards. Limitations on the size of 1D textures can be overcome by using 2D textures

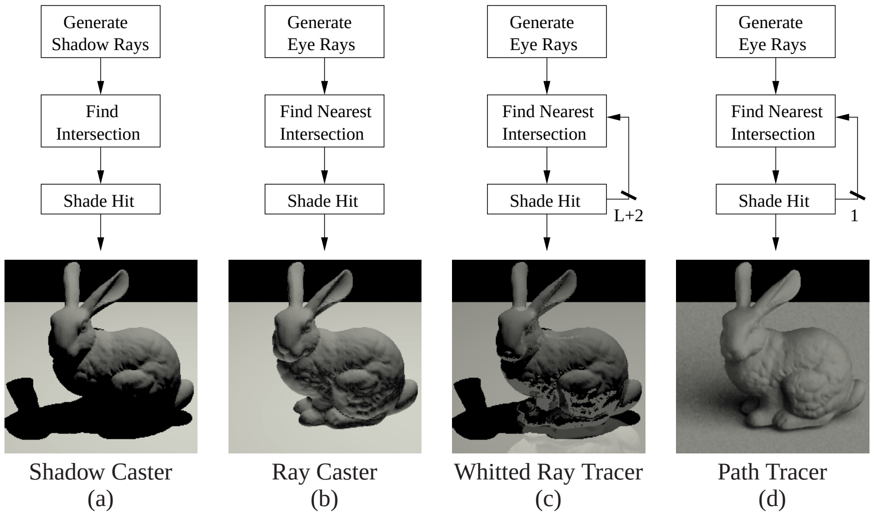

Figure 3: Data flow diagrams for the ray tracing algorithms we implement. The algorithms depicted are (a) shadow casting, (b) ray casting, (c) Whitted ray tracing, and (d) path tracing. For ray tracing, each ray-surface intersection generates $L+2$ rays, where L is the number of lights in a scene, corresponding to the number of shadow rays to be tested, and the other two are reflection and refraction rays. Path tracing randomly chooses one ray bounce to follow and the feedback path is only one ray wide.

```
float4 IntersectTriangle( float3 ro, float3 rd, int list_pos, float4 h ){
    float tri_id = texture( list_pos, trilist );
    float3 v0 = texture( tri_id, v0 );
    float3 v1 = texture( tri_id, v1 );
    float3 v2 = texture( tri_id, v2 );
    float3 edge1 = v1 - v0;
    float3 edge2 = v2 - v0;
    float3 pvec = Cross( rd, edge2 );
    float det = Dot( edge1, pvec );
    float inv_det = 1/det;
    float3 tvec = ro - v0;
    float u = Dot( tvec, pvec ) * inv_det;
    float3 qvec = Cross( tvec, edge1 );
    float v = Dot( rd, qvec ) * inv_det;
    float t = Dot( edge2, qvec ) * inv_det;
    bool validhit = select( u >= 0.0f, true, false );
    validhit = select( v >= 0, validhit, false );
    validhit = select( u+v <= 1, validhit, false );
    validhit = select( t < h[0], validhit, false );
    validhit = select( t >= 0, validhit, false );
    t = select( validhit, t, h[0] );
    u = select( validhit, u, h[1] );
    v = select( validhit, v, h[2] );
    float id = select( validhit, tri_id, h[3] );
    return float4( {t, u, v, id} );
}
```

Figure 5: Code for ray-triangle intersection.

with the proper address translation, as well as segmenting the data across multiple textures.

As with the traversal stage, the inner loop over all the triangles in a voxel involves multiple passes. Each ray requires a single pass per ray-triangle intersection.

3.1.4 Shader

The shading kernel evaluates the color contribution of a given ray at the hit point. The shading calculations are exactly like those in the standard graphics pipeline. Shading data is stored in memory much like triangle data. A set of three RGB textures, with 32-bits per channel, contains the vertex normals and vertex colors for each triangle. The hit information that is passed to the shader includes the triangle number. We access the shading information by a simple dependent texture lookup for the particular triangle specified.

By choosing different shading rays, we can implement several flavors of ray tracing using our streaming algorithm. We will look at ray casting, Whitted-style ray tracing, path tracing, and shadow casting. Figure 3 shows a simplified flow diagram for each of the methods discussed, along with an example image produced by our system.

The shading kernel optionally generates shadow, reflection, refraction, or randomly generated rays. These secondary rays are placed in texture locations for future rendering passes. Each ray is also assigned a weight, so that when it is finally terminated, its contribution to the final image may be simply added into the image [Kajiya 1986]. This technique of assigning a weight to a ray eliminates recursion and simplifies the control flow.

Ray Caster. A ray caster generates images that are identical to those generated by the standard graphics pipeline. For each pixel on the screen, an eye ray is generated. This ray is fired into the scene and returns the color of the nearest triangle it hits. No secondary rays are generated, including no shadow rays. Most previous efforts to implement interactive ray tracing have focused on this type of computation, and it will serve as our basic implementation.

Whitted Ray Tracer. The classic Whitted-style ray tracer [Whitted 1980] generates eye rays and sends them out into the scene. Upon finding a hit, the reflection model for that surface is evaluated, and then a pair of reflection and refraction rays, and a set of shadow rays – one per light source – are generated and sent out into the scene.

Path Tracer. In path tracing, rays are randomly scattered from surfaces until they hit a light source. Our path tracer emulates the Arnold renderer [Fajardo 2001]. One path is generated per sample and each path contains 2 bounces.

Shadow Caster. We simulate a hybrid system that uses the standard graphics pipeline to perform hidden surface calculation in the first pass, and then uses ray tracing algorithm to evaluate shadows. Shadow casting is useful as a replacement for shadow maps and shadow volumes. Shadow volumes can be extremely expensive to compute, while for shadow maps, it tends to be difficult to set the proper resolution. A shadow caster can be viewed as a deferred shading pass [Molnar et al. 1992]. The shadow caster pass generates shadow rays for each light source and adds that light's contribution to the final image only if no blockers are found.

Kernel	Multipass						Branching			
	Instr. Count	Memory Words			Stencil		Instr. Count	Memory Words		
		R	W	M	RS	WS		R	W	M
Generate Eye Ray	28	0	5	0	0	1	26	0	4	0
Traverse										
Setup	38	11	12	0	1	0	22	7	0	0
Step	20	14	9	1	1	1	12	0	0	1
Intersect	41	14	5	10	1	1	36	0	0	10
Shade										
Color	36	10	3	21	1	0	25	0	3	21
Shadow	16	11	8	0	1	1	10	0	0	0
Reflected	26	11	9	9	1	1	12	0	0	0
Path	17	14	9	9	1	1	11	3	0	0

Table 1: Ray tracing kernel summary. We show the number of instructions required to implement each of our kernels, along with the number of 32-bit words of memory that must be read and written between rendering passes (R, W) and the number of memory words read from random-access textures (M). Two sets of statistics are shown, one for the multipass architecture and another for the branching architecture. For the multipass architecture, we also show the number of 8-bit stencil reads (RS) and writes (WS) for each kernel. Stencil read overhead is charged for all rays, whether the kernel is executed or not.

3.2 Implementation

To evaluate the computation and bandwidth requirements of our streaming ray tracer, we implemented each kernel as an assembly language fragment program. The NVIDIA vertex program instruction set is used for fragment programs, with the addition of a few instructions as described previously. The assembly language implementation provides estimates for the number of instructions required for each kernel invocation. We also calculate the bandwidth required by each kernel; we break down the bandwidth as stream input bandwidth, stream output bandwidth, and memory (random-access read) bandwidth.

Table 1 summarizes the computation and bandwidth required for each kernel in the ray tracer, for both the multipass and the branching architectures. For the traversal and intersection kernels that involve looping, the counts for the setup and the loop body are shown separately. The branching architecture allows us to combine individual kernels together; as a result the branching kernels are slightly smaller since some initialization and termination instructions are removed. The branching architecture permits all kernels to be run together within a single rendering pass.

Using table 1, we can compute the total compute and bandwidth costs for the scene.

$$C = R * (C_r + vC_v + tC_t + sC_s) + R * P * C_{stencil}$$

Here R is the total number of rays traced. C_r is the cost to generate a ray; C_v is the cost to walk a ray through a voxel; C_t is the cost of performing a ray-triangle intersection; and C_s is the cost of shading. P is the total number of rendering passes, and $C_{stencil}$ is the cost of reading the stencil buffer. The total cost associated with each stage is determined by the number of times that kernel is invoked. This number depends on scene statistics: v is the average number of voxels pierced by a ray; t is the average number of triangles intersected by a ray; and s is the average number of shading calculations per ray. The branching architecture has no stencil buffer checks, so $C_{stencil}$ is zero. The multipass architecture must pay the stencil read cost for all rays over all rendering passes. The cost of our ray tracer on various scenes will be presented in the results section.

Finally, we present an optimization to minimize the total number of passes motivated in part by Delany's implementation of a ray tracer for the Connection Machine [Delany 1988]. The traversal and intersection kernels both involve loops. There are various strategies for nesting these loops. The simplest algorithm would be to step through voxels until any ray encounters a voxel containing triangles, and then intersect that ray with those triangles. However, this strategy would be very inefficient, since during intersection only a few rays will have encountered voxels with triangles. On a SIMD machine like the Connection Machine, this results in very low processor utilization. For graphics hardware, this yields an excessive number of passes resulting in large number of stencil read operations dominating the performance. The following is a more efficient algorithm:

```
generate eye ray
while (any(active(ray))) {
    if (oracle(ray))
        traverse(ray)
    else
        intersect(ray)
}
shade(ray)
```

After eye ray generation, the ray tracer enters a while loop which tests whether any rays are *active*. Active rays require either further traversals or intersections; inactive rays have either hit triangles or traversed the entire grid. Before each pass, an oracle is called. The oracle chooses whether to run a traverse or an intersect pass. Various oracles are possible. The simple algorithm above runs an intersect pass if *any* rays require intersection tests. A better oracle, first proposed by Delany, is to choose the pass which will perform the most work. This can be done by calculating the percentage of rays requiring intersection vs. traversal. In our experiments, we found that performing intersections once 20% of the rays require intersection tests produced the minimal number of passes, and is within a factor of two to three of optimal for a SIMD algorithm performing a single computation per rendering pass.

To implement this oracle, we assume graphics hardware maintains a small set of counters over the stencil buffer, which contains the state of each ray. A total of eight counters (one per stencil bit) would be more than sufficient for our needs since we only have four states. Alternatively, we could use the OpenGL histogram operation for the oracle if this operation were to be implemented with high performance for the stencil buffer.

4 Results

4.1 Methodology

We have implemented functional simulators of our streaming ray tracer for both the multipass and branching architectures. These simulators are high level simulations of the architectures, written in the C++ programming language. These simulators compute images and gather scene statistics. Example statistics include the average number of traversal steps taken per ray, or the average number of

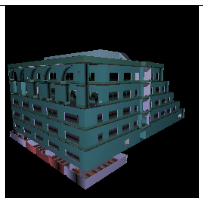			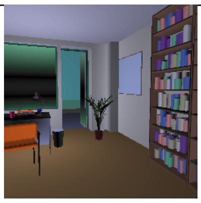			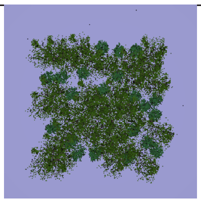						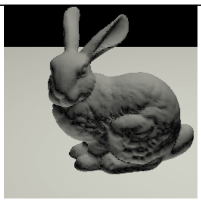		
Soda Hall Outside			Soda Hall Inside			Forest Top Down			Forest Inside			Bunny Ray Cast		
v	t	s	v	t	s	v	t	s	v	t	s	v	t	s
14.41	2.52	0.44	26.11	40.46	1.00	81.29	34.07	0.96	130.7	47.90	0.97	93.93	13.88	0.82

Figure 6: Fundamental scene statistics for our test scenes. The statistics shown match the cost model formula presented in section 3.2. Recall that v is the average number of voxels pierced by a ray; t is the average number of triangles intersected by a ray; and s is the average number of shading calculations per ray. Soda hall has 1.5M triangles, the forest has 1.0M triangles, and the Stanford bunny has 70K triangles. All scenes are rendered at 1024x1024 pixels.

ray-triangle intersection tests performed per ray. The multipass architecture simulator also tracks the number and type of rendering passes performed, as well as stencil buffer activity. These statistics allow us to compute the cost for rendering a scene by using the cost model described in section 3.

Both the multipass and the branching architecture simulators generate a trace file of the memory reference stream for processing by our texture cache simulator. In our cache simulations we used a 64KB direct-mapped texture cache with a 48-byte line size. This line size holds four floating point RGB texels, or three floating point RGBA texels with no wasted space. The execution order of fragment programs effects the caching behavior. We execute kernels as though there were a single pixel wide graphics pipeline. It is likely that a GPU implementation will include multiple parallel fragment pipelines executing concurrently, and thus their accesses will be interleaved. Our architectures are not specified at that level of detail, and we are therefore not able to take such effects into account in our cache simulator.

We analyze the performance of our ray tracer on five viewpoints from three different scenes, shown in figure 6.

- Soda Hall is a relatively complex model that has been used to evaluate other real-time ray tracing systems [Wald et al. 2001b]. It has walls made of large polygons and furnishings made from very small polygons. This scene has high depth complexity.
- The forest scene includes trees with millions of tiny triangles. This scene has moderate depth complexity, and it is difficult to perform occlusion culling. We analyze the cache behavior of shadow and reflection rays using this scene.
- The bunny was chosen to demonstrate the extension of our ray tracer to support shadows, reflections, and path tracing.

Figure 7 shows the computation and bandwidth requirements of our test scenes. The computation and bandwidth utilized is broken down by kernel. These graphs clearly show that the computation and bandwidth for both architectures is dominated by grid traversal and triangle intersection.

Choosing an optimal grid resolution for scenes is difficult. A finer grid yields fewer ray-triangle intersection tests, but leads to more traversal steps. A coarser grid reduces the number of traversal steps, but increases the number of ray-triangle intersection tests. We attempt to keep voxels near cubical shape, and specify grid resolution by the minimal grid dimension acceptable along any dimension of the scene bounding box. For the bunny, our minimal grid dimension is 64, yielding a final resolution of $98 \times 64 \times 163$. For the larger Soda Hall and forest models, this minimal dimension is 128, yielding grid resolutions of $250 \times 198 \times 128$ and $581 \times 128 \times 581$ respectively. These resolutions allow our algorithms to spend equal amounts of time in the traversal and intersection kernels.

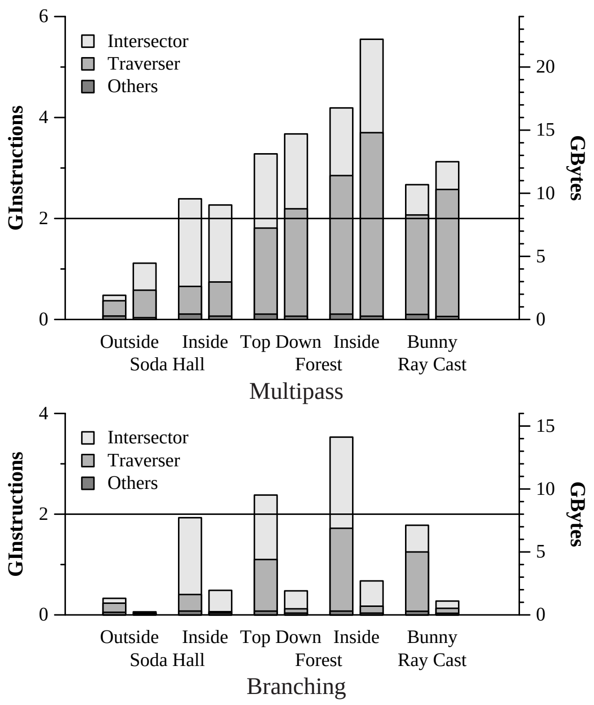

Figure 7: Compute and bandwidth usage for our scenes. Each column shows the contribution from each kernel. Left bar on each plot is compute, right is bandwidth. The horizontal line represents the per-second bandwidth and compute performance of our hypothetical architecture. All scenes were rendered at 1024×1024 pixels.

4.2 Architectural Comparisons

We now compare the multipass and branching architectures. First, we investigate the implementation of the ray caster on the multipass architecture. Table 2 shows the total number of rendering passes and the distribution of passes amongst the various kernels. The total number of passes varies between 1000-3000. Although the number of passes seems high, this is the total number needed to render the scene. In the conventional graphics pipeline, many fewer passes per object are used, but many more objects are drawn. In our system, each pass only draws a single rectangle, so the speed of the geometry processing part of the pipeline is not a factor.

We also evaluate the efficiency of the multipass algorithm. Recall that rays may be traversing, intersecting, shading, or done. The efficiency of a pass depends on the percentage of rays processed in that pass. In these scenes, the efficiency is between 6-10% for all of the test scenes except for the outside view of Soda Hall. This

	Pass Breakdown				Per Ray Maximum		SIMD
	Total	Traversal	Intersection	Other	Traversals	Intersections	Efficiency
Soda Hall Outside	2443	692	1747	4	384	1123	0.009
Soda Hall Inside	1198	70	1124	4	60	1039	0.061
Forest Top Down	1999	311	1684	4	137	1435	0.062
Forest Inside	2835	1363	1468	4	898	990	0.068
Bunny Ray Cast	1085	610	471	4	221	328	0.105

Table 2: Breakdown of passes in the multipass system. Intersection and traversal make up the bulk of passes in the systems, with the rest of the passes coming from ray generation, traversal setup, and shading. We also show the maximum number of traversal steps and intersection tests for per ray. Finally, SIMD efficiency measures the average fraction of rays doing useful work for any given pass.

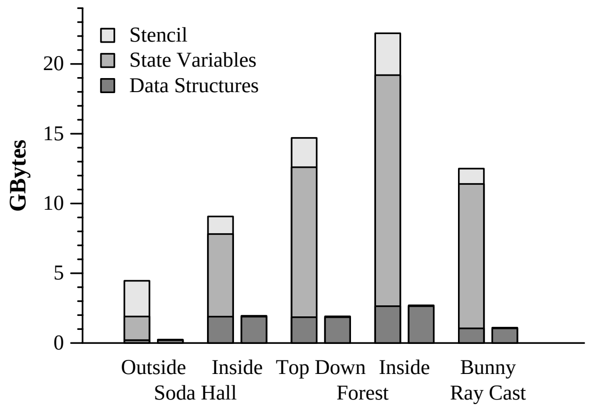

Figure 8: Bandwidth consumption by data type. Left bars are for multipass, right bars for branching. Overhead for reading the 8-bit stencil value is shown on top. State variables are data written to and read from texture between passes. Data structure bandwidth comes from read-only data: triangles, triangle lists, grid cells, and shading data. All scenes were rendered at 1024 × 1024 pixels.

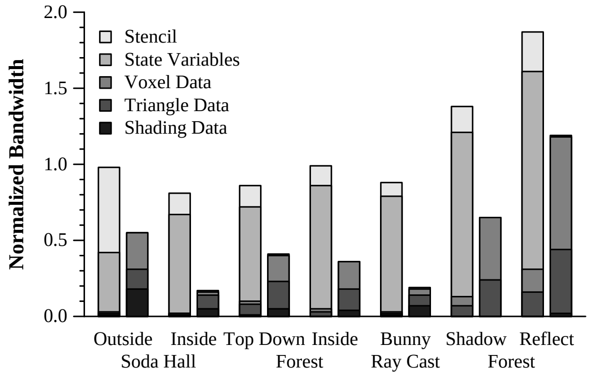

Figure 9: Ratio of bandwidth with a texture cache to bandwidth without a texture cache. Left bars are for multipass, right bars for branching. Within each bar, the bandwidth consumed with a texture cache is broken down by data type. All scenes were rendered at 1024 × 1024 pixels.

viewpoint contains several rays that miss the scene bounding box entirely. As expected, the resulting efficiency is much lower since these rays never do any useful work during the rest of the computation. Although 10% efficiency may seem low, the fragment processor utilization is much higher because we are using early fragment kill to avoid consuming compute resources and non-stencil bandwidth for the fragment. Finally, table 2 shows the maximum number of traversal steps and intersection tests that are performed per ray. Since the total number of passes depends on the worst case ray, these numbers provide lower bounds on the number of passes needed. Our multipass algorithm interleaves traversal and intersection passes and comes within a factor of two to three of the optimal number of rendering passes. The naive algorithm, which performs an intersection as soon as any ray hits a full voxel, requires at least a factor of five times more passes than optimal on these scenes.

We are now ready to compare the computation and bandwidth requirements of our test scenes on the two architectures. Figure 8 shows the same bandwidth measurements shown in figure 7 broken down by data type instead of by kernel. The graph shows that, as expected, all of the bandwidth required by the branching architecture is for reading voxel and triangle data structures from memory. The multipass architecture, conversely, uses most of its bandwidth for writing and reading intermediate values to and from texture memory between passes. Similarly, saving and restoring these intermediates requires extra instructions. In addition, significant bandwidth is devoted to reading the stencil buffer. This extra computation and bandwidth consumption is the fundamental limitation of the multipass algorithm.

One way to reduce both the number of rendering passes and the bandwidth consumed by intermediate values in the multipass architecture is to unroll the inner loops. We have presented data for a single traversal step or a single intersection test performed per ray in a rendering pass. If we instead unroll our kernels to perform four traversal steps or two intersection tests, all of our test scenes reduce their total bandwidth usage by 50%. If we assume we can suppress triangle and voxel memory references if a ray finishes in the middle of the pass, the total bandwidth reduction reaches 60%. At the same time, the total instruction count required to render each scene increases by less than 10%. With more aggressive loop unrolling the bandwidth savings continue, but the total instruction count increase varies by a factor of two or more between our scenes. These results indicate that loop unrolling can make up for some of the overhead inherent in the multipass architecture, but unrolling does not achieve the compute to bandwidth ratio obtained by the branching architecture.

Finally, we compare the caching behavior of the two implementations. Figure 9 shows the bandwidth requirements when a texture cache is used. The bandwidth consumption is normalized by dividing by the non-caching bandwidth reported earlier. Inspecting this graph we see that the multipass system does not benefit very much from texture caching. Most of the bandwidth is being used for streaming data, in particular, for either the stencil buffer or for intermediate results. Since this data is unique to each kernel invocation, there is no reuse. In contrast, the branching architecture utilizes the texture cache effectively. Since most of its bandwidth is devoted to reading shared data structures, there is reuse. Studying the caching behavior of triangle data only, we see that a 96-99% hit rate is achieved by both the multipass and the branching system. This high hit rate suggests that triangle data caches well, and that we have a fairly small working set size.

In summary, the implementation of the ray caster on the multipass architecture has achieved a very good balance between computation and bandwidth. The ratio of instruction count to bandwidth matches the capabilities of a modern GPU. For example, the

Extension	Relative Instructions	Relative Bandwidth
Shadow Caster	0.85	1.15
Whitted Ray Tracer	2.62	3.00
Path Tracer	3.24	4.06

Table 3: Number of instructions and amount of bandwidth consumed by the extended algorithms to render the bunny scene using the branching architecture, normalized by the ray casting cost.

NVIDIA GeForce3 is able to execute approximately 2G instructions/s in its fragment processor, and has roughly 8GB/s of memory bandwidth. Expanding the traversal and intersection kernels to perform multiple traversal steps or intersection tests per pass reduces the bandwidth required for the scene at the cost of increasing the computational requirements. The amount of loop unrolling can be changed to match the computation and bandwidth capabilities of the underlying hardware. In comparison, the branching architecture consumes fewer instructions and significantly less bandwidth. As a result, the branching architecture is severely compute-limited based on today's GPU bandwidth and compute rates. However, the branching architecture will become more attractive in the future as the compute to bandwidth ratio on graphics chips increases with the introduction of more parallel fragment pipelines.

4.3 Extended Algorithms

With an efficient ray caster in place, implementing extensions such as shadow casting, full Whitted ray tracing, or path tracing is quite simple. Each method utilizes the same ray-triangle intersection loop we have analyzed with the ray caster, but implements a different shading kernel which generates new rays to be fed back through our system. Figure 3 shows images of the bunny produced by our system for each of the ray casting extensions we simulate. The total cost of rendering a scene depends on both the number of rays traced and the cache performance.

Table 3 shows the number of instructions and bandwidth required to produce each image of the bunny relative to the ray casting cost, all using the branching architecture. The path traced bunny was rendered at 256×256 pixels with 64 samples and 2 bounces per pixel while the others were rendered at 1024×1024 pixels. The ray cast bunny finds a valid hit for 82% of its pixels and hence 82% of the primary rays generate secondary rays. If all rays were equal, one would expect the shadow caster to consume 82% of the instructions and bandwidth of the ray caster; likewise the path tracer would consume 3.2 times that of the ray caster. Note that the instruction usage is very close to the expected value, but that the bandwidth consumed is more.

Additionally, secondary rays do not cache as well as eye rays, due to their generally incoherent nature. The last two columns of figure 9 illustrate the cache effectiveness on secondary rays, measured separately from primary rays. For these tests, we render the inside forest scene in two different styles. "Shadow" is rendered with three light sources with each hit producing three shadow rays. "Reflect" applies a two bounce reflection and single light source shading model to each primitive in the scene. For the multipass rendering system, the texture cache is unable to reduce the total bandwidth consumed by the system. Once again the streaming data destroys any locality present in the triangle and voxel data. The branching architecture results demonstrate that scenes with secondary rays can benefit from caching. The system achieves a 35% bandwidth reduction for the shadow computation. However caching for the reflective forest does not reduce the required bandwidth. We are currently investigating ways to improve the performance of our system for secondary rays.

5 Discussion

In this section, we discuss limitations of the current system and future work.

5.1 Acceleration Data Structures

A major limitation of our system is that we rely on a preprocessing step to build the grid. Many applications contain dynamic geometry, and to support these applications we need fast incremental updates to the grid. Building acceleration data structures for dynamic scenes is an active area of research [Reinhard et al. 2000]. An interesting possibility would be to use graphics hardware to build the acceleration data structure. The graphics hardware could "scan convert" the geometry into a grid. However, the architectures we have studied in this paper cannot do this efficiently; to do operations like rasterization within the fragment processor they would need the ability to write to arbitrary memory locations. This is a classic scatter operation and would move the hardware even closer to a general stream processor.

In this research we assumed a uniform grid. Uniform grids, however, may fail for scenes containing geometry and empty space at many levels of detail. Since we view texture memory as random-access memory, hierarchical grids could be added to our system.

Currently graphics boards contain relatively small amounts of memory (in 2001 a typical board contains 64MB). Some of the scenes we have looked at require 200MB - 300MB of texture memory to store the scene. An interesting direction for future work would be to study hierarchical caching of the geometry as is commonly done for textures. The trend towards unified system and graphics memory may ultimately eliminate this problem.

5.2 CPU vs. GPU

Wald et al. have developed an optimized ray tracer for a PC with SIMD floating point extensions [Wald et al. 2001b]. On an 800 MHz Pentium III, they report a ray-triangle intersection rate of 20M intersections/s. Carr et al. [2002] achieve 114M ray-triangle intersections/s on an ATI Radeon 8500 using limited fixed point precision. Assuming our proposed hardware ran at the same speed as a GeForce3 (2G instructions/s), we could compute 56M ray-triangle intersections/s. Our branching architecture is compute limited; if we increase the instruction issue rate by a factor of four (8G instructions/s) then we would still not use all the bandwidth available on a GeForce3 (8GB/s). This would allow us to compute 222M ray-triangle intersections per second. We believe because of the inherently parallel nature of fragment programs, the number of GPU instructions that can be executed per second will increase much faster than the number of CPU SIMD instructions.

Once the basic feasibility of ray tracing on a GPU has been demonstrated, it is interesting to consider modifications to the GPU that support ray tracing more efficiently. Many possibilities immediately suggest themselves. Since rays are streamed through the system, it would be more efficient to store them in a stream buffer than a texture map. This would eliminate the need for a stencil buffer to control conditional execution. Stream buffers are quite similar to F-buffers which have other uses in multipass rendering [Mark and Proudfoot 2001]. Our current implementation of the grid traversal code does not map well to the vertex program instruction set, and is thus quite inefficient. Since grid traversal is so similar to rasterization, it might be possible to modify the rasterizer to walk through the grid. Finally, the vertex program instruction set could be optimized so that ray-triangle intersection could be performed in fewer instructions.

Carr et al. [2002] have independently developed a method of using the GPU to accelerate ray tracing. In their system the GPU is only used to accelerate ray-triangle intersection tests. As in our system, GPU memory is used to hold the state for many active rays. In their system each triangle in turn is fed into the GPU and tested for intersection with all the active rays. Our system differs from theirs in that we store all the scene triangles in a 3D grid on the GPU; theirs stores the acceleration structure on the CPU. We also run the entire ray tracer on the GPU. Our system is much more efficient than theirs since we eliminate the GPU-CPU communication bottleneck.

5.3 Tiled Rendering

In the multipass architecture, the majority of the memory bandwidth was consumed by saving and restoring temporary variables. Since these streaming temporaries are only used once, there is no bandwidth savings due to the cache. Unfortunately, when these streaming variables are accessed as texture, they displace cacheable data structures. The size of the cache we used is not large enough to store the working set if it includes both temporary variables and data structures. The best way to deal with this problem is to separate streaming variables from cacheable variables.

Another solution to this problem is to break the image into small tiles. Each tile is rendered to completion before proceeding to the next tile. Tiling reduces the working set size, and if the tile size is chosen so that the working set fits into the cache, then the streaming variables will not displace the cacheable data structures. We have performed some preliminary experiments along these lines and the results are encouraging.

6 Conclusions

We have shown how viewing a programmable graphics processor as a general parallel computation device can help us leverage the graphics processor performance curve and apply it to more general parallel computations, specifically ray tracing. We have shown that ray casting can be done efficiently in graphics hardware. We hope to encourage graphics hardware to evolve toward a more general programmable stream architecture.

While many believe a fundamentally different architecture would be required for real-time ray tracing in hardware, this work demonstrates that a gradual convergence between ray tracing and the feed-forward hardware pipeline is possible.

7 Acknowledgments

We would like to thank everyone in the Stanford Graphics Lab for contributing ideas to this work. We thank Matt Papakipos from NVIDIA for his thoughts on next generation graphics hardware, and Kurt Akeley and our reviewers for their comments. Katie Tillman stayed late and helped with editing. This work was sponsored by DARPA (contracts DABT63-95-C-0085 and MDA904-98-C-A933), ATI, NVIDIA, Sony, and Sun.

References

3DLABS, 2001. OpenGL 2.0 whitepapers web site. http://www.3dlabs.com/support/developer/ogl2/index.htm.

ALVERSON, R., CALLAHAN, D., CUMMINGS, D., KOBLENZ, B., PORTERFIELD, A., AND SMITH, B. 1990. The Tera computer system. In *Proceedings of the 1990 International Conference on Supercomputing*, 1–6.

AMANATIDES, J., AND WOO, A. 1987. A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, 3–10.

ANDERSON, B., STEWART, A., MACAULAY, R., AND WHITTED, T. 1997. Accommodating memory latency in a low-cost rasterizer. In *1997 SIGGRAPH / Eurographics Workshop on Graphics hardware*, 97–102.

ATI, 2001. RADEON 8500 product web site. http://www.ati.com/products/pc/radeon8500128/index.html.

CARR, N. A., HALL, J. D., AND HART, J. C. 2002. The ray engine. Tech. Rep. UIUCDCS-R-2002-2269, Department of Computer Science, University of Illinois.

DELANY, H. C. 1988. Ray tracing on a connection machine. In *Proceedings of the 1988 International Conference on Supercomputing*, 659–667.

FAJARDO, M. 2001. Monte carlo ray tracing in action. In *State of the Art in Monte Carlo Ray Tracing for Realistic Image Synthesis - SIGGRAPH 2001 Course 29*. 151–162.

FUJIMOTO, A., TANAKA, T., AND IWATA, K. 1986. ARTS: Accelerated ray tracing system. *IEEE Computer Graphics and Applications 6*, 4, 16–26.

HALL, D., 2001. The AR350: Today's ray trace rendering processor. 2001 SIGGRAPH / Eurographics Workshop On Graphics Hardware - Hot 3D Session 1. http://graphicshardware.org/previous/www_2001/presentations/ Hot3D_Daniel_Hall.pdf.

HAVRAN, V., PRIKRYL, J., AND PURGATHOFER, W. 2000. Statistical comparison of ray-shooting efficiency schemes. Tech. Rep. TR-186-2-00-14, Institute of Computer Graphics, Vienna University of Technology.

IGEHY, H., ELDRIDGE, M., AND PROUDFOOT, K. 1998. Prefetching in a texture cache architecture. In *1998 SIGGRAPH / Eurographics Workshop on Graphics hardware*, 133–ff.

KAJIYA, J. T. 1986. The rendering equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH 86)*, 143–150.

KHAILANY, B., DALLY, W. J., RIXNER, S., KAPASI, U. J., MATTSON, P., NAMKOONG, J., OWENS, J. D., AND TOWLES, B. 2000. IMAGINE: Signal and image processing using streams. In *Hot Chips 12*. IEEE Computer Society Press.

KIRK, D., 2001. GeForce3 architecture overview. http://developer.nvidia.com/docs/IO/1271/ATT/GF3ArchitectureOverview.ppt.

LINDHOLM, E., KILGARD, M. J., AND MORETON, H. 2001. A user-programmable vertex engine. In *Proceedings of ACM SIGGRAPH 2001*, 149–158.

MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: A rasterization-order FIFO buffer for multi-pass rendering. In *2001 SIGGRAPH / Eurographics Workshop on Graphics Hardware*.

MARSHALL, B., 2001. DirectX graphics future. Meltdown 2001 Conference. http://www.microsoft.com/mscorp/corpevents/meltdown2001/ppt/DXG9.ppt.

MICROSOFT, 2001. DirectX product web site. http://www.microsoft.com/directx/.

MOLNAR, S., EYLES, J., AND POULTON, J. 1992. PixelFlow: High-speed rendering using image composition. In *Computer Graphics (Proceedings of ACM SIGGRAPH 92)*, 231–240.

NVIDIA, 2001. GeForce3 Ti Family: Product overview. 10.01v1. http://www.nvidia.com/docs/lo/1050/SUPP/gf3ti_overview.pdf.

PARKER, S., SHIRLEY, P., LIVNAT, Y., HANSEN, C., AND SLOAN, P.-P. 1998. Interactive ray tracing for isosurface rendering. In *IEEE Visualization '98*, 233–238.

PARKER, S., MARTIN, W., SLOAN, P.-P. J., SHIRLEY, P., SMITS, B., AND HANSEN, C. 1999. Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, 119–126.

PEERCY, M. S., OLANO, M., AIREY, J., AND UNGAR, P. J. 2000. Interactive multi-pass programmable shading. In *Proceedings of ACM SIGGRAPH 2000*, 425–432.

REINHARD, E., SMITS, B., AND HANSEN, C. 2000. Dynamic acceleration structures for interactive ray tracing. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, 299–306.

SPITZER, J., 2001. Texture compositing with register combiners. http://developer.nvidia.com/docs/IO/1382/ATT/RegisterCombiners.pdf.

TORBORG, J., AND KAJIYA, J. T. 1996. Talisman: Commodity realtime 3D graphics for the PC. In *Proceedings of ACM SIGGRAPH 96*, 353–363.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, 277–288.

WALD, I., SLUSALLEK, P., BENTHIN, C., AND WAGNER, M. 2001. Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*, 3, 153–164.

WHITTED, T. 1980. An improved illumination model for shaded display. *Communications of the ACM 23*, 6, 343–349.