A High-Speed Ray Tracing Engine Built on a Field-Programmable System

Joshua Fender
University of Toronto
fender@eecg.utoronto.ca

Jonathan Rose
University of Toronto
jayar@eecg.utoronto.ca

Abstract

Ray tracing is a method of rendering high-quality images and video by calculating what happens to virtual light rays in a 3-dimensional scene. It is capable of creating far more realism than traditional Z-buffering methods. This paper describes the design of a hardware ray tracing system implemented on a multi-FPGA Xilinx Virtex-E prototyping system. The result is a hardware ray tracer that is capable of out-performing a 2.4GHz Pentium 4, running a well-known high performance software ray tracing algorithm, by up to a factor of thirty. When these results are projected forward into a next generation FPGA system, consisting of a single large Virtex 2 Pro FPGA, it is found that the system should be able to out perform the same Pentium 4 by up to two orders of magnitude, and the fastest known hardware implementation, the AR350, by up to a factor of three.

1. Introduction

This paper describes a hardware ray tracing engine that was implemented using a field-programmable rapid prototyping system. Ray tracing [1] is a method of rendering high-quality images by calculating what happens to the light rays in a 3-dimensional scene. It is capable of creating far more realism than traditional Z-buffering methods [2].

To date, raster graphics has dominated ray tracing approaches because ray tracing requires a much larger amount of computation. It is relevant to note, however, that the algorithmic complexity of ray tracing is logarithmic in the scene size, while raster is typically linear. Ray tracing currently requires more computation because the constants in front of the logarithm are large; ultimately its logarithmic complexity will win out, for large scenes, and so it is an avenue worth pursuing for this reason alone.

Several different hardware approaches have been tried in the past to accelerate ray tracing to a point where they are competitive with raster graphics. These include several general purpose renderers [3],[4],[5] as well as a number of application specific renderers, such as [6] and [7]. The approaches implemented the hardware as custom ASICs to achieve maximum performance, but this caused high development costs high and successive versions to be far apart. A more flexible and evolvable approach is to target programmable hardware and iteratively improve the designs, which we do in the present work.

We present a FPGA ray tracing architecture that is capable of exceeding the performance levels currently available from custom ray tracing hardware. This goal is achieved in two steps. The first step involves designing a simple prototype ray tracer and using this system to discover the various issues that affect a hardware ray tracer's performance. The second phase then involves using these known issues to design an enhanced ray tracing system and evaluate its projected performance.

This paper is divided into seven sections: Section 2 presents the basics of ray tracing, Section 3 describes the multi-FPGA rapid prototyping system used, Section 4 describes the prototype system, Section 5 describes performance results of this system, Section 6 examines the various trade offs involved in the design of the enhanced system, and Section 7 presents the performance of the enhanced system.

2. Ray Tracing Basics

Conventional rasterized rendering is an object-centric algorithm. That is, each object is processed in turn and rendered into the frame buffer. A ray tracing algorithm uses a pixel-centric view of renderering by attempting to determine which object should be visible for a given pixel.

Figure 1 illustrates how this pixel-based algorithm works. For each pixel of the projection plane *P*, a ray *R* is generated from the eye point through the pixel. This ray is then intersected with the 3D scene to determine which object is visible to the viewer through that pixel. Once the visible object is found, various models are used to determine the pixel's final colour. This process is then repeated for every pixel in the scene and the image is rendered.

The core algorithm in ray tracing is the one that determines which object in the 3D scene is struck by a given ray. For the purposes of this paper, scenes will be constrained to only those that consist of triangle objects. This allows for the use of the very simple intersection algorithm that is described in the following section.

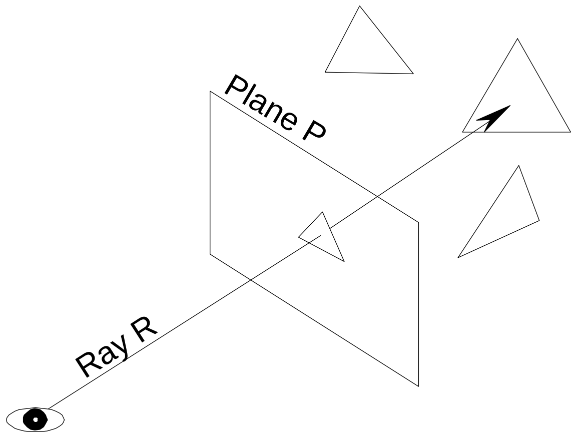

Figure 1. A single ray from eye through screen to triangle in scene

2.1. Ray-Triangle Intersection

There are a number of different algorithms that can calculate the intersection point between a ray and a triangle. Of these they are three major variants: those that use the plane equation [8], those that use 6D Plücker space [9], and those that use barycentric coordinates [10]. After evaluating the various algorithms it was found that the barycentric coordinate method was the most computationally efficient when implemented in hardware.

The algorithm is as follows:

Given a parameterized ray $\vec{R}(t)$ with direction $\vec{D}$ and origin $\vec{O}$,

$$\vec{R}(t) = \vec{O} + t\vec{D} \quad where \quad \vec{R}(t), \vec{O}, \vec{D} \in R^3 \qquad (1)$$

and a triangle with vertices $\vec{V}_0, \vec{V}_1$, and $\vec{V}_2$, barycentric coordinates can be used to determine if the two intersect. Barycentric coordinates provided a new coordinate space, *(u, v)*, which includes the set of all points that lay in the plane of a given triangle. Equation (2) shows the relationship between R^3 and barycentric space.

$$\vec{T}(u,v) = (1-u-v)\vec{V}_0 + u\vec{V}_1 + v\vec{V}_2 \qquad (2)$$

It can be shown that a given point in barycentric space is contained within the triangle if, and only if, the conditions on *(u, v)*, shown below, are met.

$$u \ge 0, v \ge 0, \ u + v \le 1 \qquad (3)$$

By equating equations (1) and (2), as shown in equation (4), we can determine if there is an intersection between a given ray and a given triangle. If the conditions shown in equation (3) are met, then the ray and triangle intersect.

$$\begin{bmatrix} -\vec{D} & \vec{E}_1 & \vec{E}_2 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \vec{T} \quad where \quad \begin{matrix} \vec{E}_1 = \vec{V}_1 - \vec{V}_0 \\ \vec{E}_2 = \vec{V}_2 - \vec{V}_0 \\ \vec{T} = \vec{O} - \vec{V}_0 \end{matrix} \qquad (4)$$

This equation can be further simplified through the use of Cramer's rule:

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{\vec{P} \cdot \vec{E}_1} \begin{bmatrix} \vec{Q} \cdot \vec{E}_2 \\ \vec{P} \cdot \vec{T} \\ \vec{Q} \cdot \vec{D} \end{bmatrix} \quad where \quad \begin{matrix} \vec{P} = \vec{D} \times \vec{E}_2 \\ \vec{Q} = \vec{T} \times \vec{E}_1 \end{matrix} \qquad (5)$$

It is this equation, in combination with the simple test for intersection of equation (3), which will be implemented in hardware.

2.2. A Hierarchical Data Structure for Reducing Algorithmic Complexity

The algorithm described in the previous section intersects a ray with only a single triangle. To render a scene it is necessary to intersect a ray with the entire scene of triangles, of which there are typically many tens of thousands. The brute force approach is to test every object in the scene and find the nearest intersection. This method results in a very simple algorithm with linear time complexity.

It is possible to reduce this complexity by hierarchically subdividing the set of objects based on their position in space. This allows the culling of a large number of objects with a small number of simple tests. Figure 2 shows an example of one possible hierarchy. The large dashed boxes represent the first level of the hierarchy. Through a simple test between the ray and the two boxes, it is easy to see that only the boxes contained within box *B* needs to be tested as box *A* is missed entirely. This process can then recursively applied to find that only box *C* in the second level of hierarchy needs to be tested. In this example three quarters of the scene is eliminated with only 4 simple tests. It is through this hierarchy that logarithmic complexity can be achieved.

The actual hierarchical structure implemented in the prototype system is a three level bounding hierarchy where the bounding objects can be defined with 6

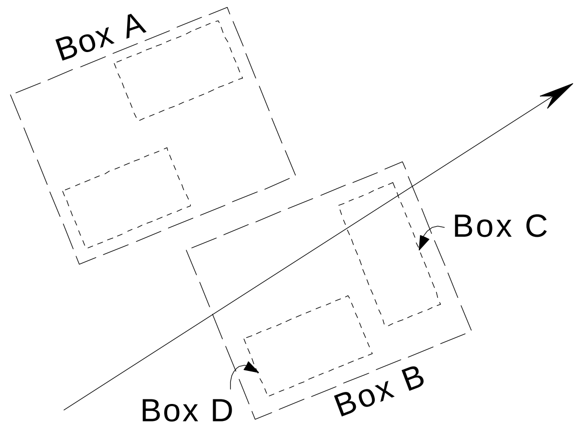

Figure 2. A sample bounding box hierarchy

arbitrary quadrilaterals. This structure was selected as it can be implemented by wrapping the existing ray triangle intersection hardware with control logic. Other hierarchy options, such as BSP trees [11], or octrees [12], would require additional dedicated datapath logic.

3. The Transmogrifier-3 Prototyping System

The hardware ray tracer described in this paper was developed using the Transmogrifier-3a [13] rapid prototyping system. The Transmogrifier-3a system consists of two separate components, the hardware system and a software tool flow.

The hardware system contains four Xilinx Virtex 2000E FPGAs, each attached to two megabytes of external SRAM. These FPGAs are connected to a host computer through a custom PCI interface that allows for both device programming and interaction with the circuit being tested. The FPGAs are also connected to each other to allow inter-chip communication.

The software tool flow used to implement our system consists of Synplicity's Synplify Pro synthesizer and Xilinx's standard place and route tool. The automated pipelining and register balancing functions of Synplify Pro were used to increase performance without time-consuming hand optimizations.

The Transmogrifier-3 system has been used to successfully implement both Stereo Vision [14] and Genome processing applications [15].

4. Prototype Ray Tracing System

Figure 3 illustrates the top-level of the ray tracing processor. The system is composed of two separate components, each implemented on different FPGAs in the Transmogrifier-3a. FPGA 0 contains the ray triangle intersection unit, which implements Möller's barycentric algorithm as described in Section 2.1, and FPGA 1 contains the control logic for implementing the bounding box hierarchy algorithm discussed in Section 2.2.

The following sections describe the data representation used to describe a 3D scene, the functionality of the ray triangle intersection unit, and the functionality of the bounding hierarchy controller.

4.1. Data Representation

Software ray tracers typically use floating-point calculations, but in hardware floating point arithmetic is tremendously expensive in area. Instead, we use a fixed-point number representation that maintains sufficient intermediate precision in all calculations.

The selection of the fixed-point data widths was guided by various hardware constraints. The memory on the

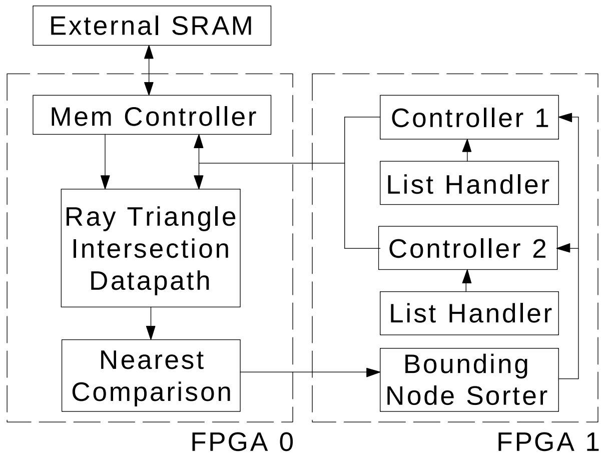

Figure 3. Architecture of ray tracing processor

Transmogrifier-3a runs at a maximum clock speed of 50MHz,and so the ray tracing processor is synchronized to this speed. Our experience with the Xilinx Virtex E suggests that only arithmetic operations that have results of 64bits or less can complete in one clock cycle.

By using this constraint and working backwards through Möller's algorithm it can be found that a working 3D space of 28bits can be used, provided that a limit is applied to the maximum triangle size. A triangle can be defined by a 28bit origin and 16bits describing the position of the other two vertices relative to the 28bit origin. This allows for the triangle to be placed anywhere in 3D space but to have a limited maximum size. In real scenes this restriction could cause problems as often a scene contains objects that are of vastly different scales, such as a piece of popcorn in the middle of a stadium.

To address the problem of objects that are of vastly different scales two additional bits can be added to optionally scale the relative offsets. Effectively these two bits allow the edge's granularity to be scaled by 1, 16, 256, or 4096, which result in the maximum edge being able to span the entire 28 bit space. Conveniently, this pseudo-floating point representation can be implemented using only a shift to the right of the resulting *(u,v)* coordinates of Möller's algorithm.

4.2. Hardware Ray-Triangle Intersection Unit

The function of the ray triangle intersection unit is to receive a list of both triangle identifiers (which are memory pointers to the triangle vertices) and rays, and then to return the identifier of the first triangle that is intersected by each ray. This unit is capable of reading one triangle every three cycles and intersecting it with three different rays.

The ray-triangle intersection unit is composed of three major components: the memory controller, the ray-triangle intersection datapath, and the nearest comparison unit.

The memory controller reads the requested triangle data from memory, the datapath intersects the triangle with a given ray, and the nearest comparison unit compares the different intersection results to determine which is closest.

4.2.1. Ray-Triangle Intersection Datapath

The ray-triangle intersection datapath is responsible for receiving a list of triangles from the memory controller, and intersecting each with three different rays. The results of these intersections are then passed on to the nearest comparison unit for later tabulation. The ratio of comparing 3 rays per triangle was necessary to match the memory speed to the computation speed: the memory controller requires three clock cycles to read a triangle, where as the datapath can perform an intersection test every cycle. By intersecting three rays with every triangle we hide this memory access time.

The datapath utilizes the maximum amount of algorithmic parallelism and is completely unwrapped to maximize throughput. For example, the cross product units (describe in Section 2.1, equation 5) each contain 6 multipliers and 3 adders running in parallel. In total, the pipeline requires 7 cycles to determine if a ray strikes a triangle and another 31 cycles to perform the division necessary to find the actual intersection point. The division units do not necessarily need to be contained in the pipeline as they are only required once an intersected triangle is found. In a system with multiple pipelines these units could be shared to save area.

The initial 7 pipeline stages use approximately 20,000 4 input look-up tables, or LUTs, to implement, and the divisions an additional 8,000 LUTs. In addition to this over 11,000 flip-flops are also required. The entire pipeline runs at 50Mhz on the Xilinx Virtex 2000E.

4.2.2. Nearest Comparison Unit

The ultimate goal of the ray triangle intersection unit is to return the nearest triangle that is intersected by a given ray, is the location of the intersection point, and is the distance from the eye to the intersection point. The nearest comparison unit tabulates the individual results from the datapath and determines the intersected triangle that is closest to the "eye", as this is the one that will be "seen". Figure 4 illustrates this circuit.

The unit contains registers with the triangle ID and distance of the nearest intersection point currently found. Each new intersection point is compared against this registered version and if the new intersection point is found to be closer it is stored and used for future comparisons.

The nearest comparison unit requires 35 LUTs and 88 flip-flops.

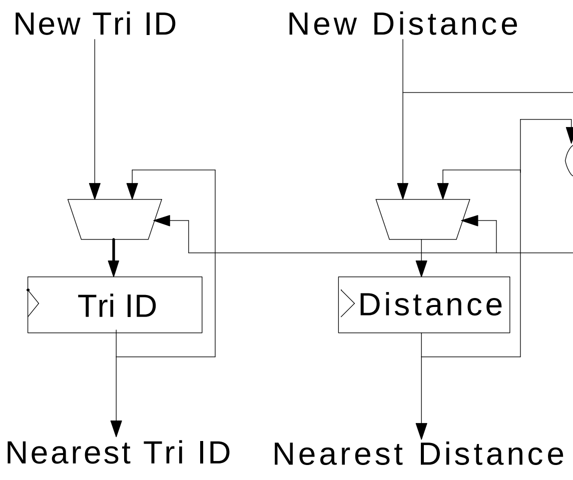

Figure 4. Nearest comparison unit

4.3. Bounding Hierarchy Unit

The ray triangle intersection unit provides the fundamental function of a ray tracer but lacks the intelligence to implement the more complicated bounding box hierarchy acceleration method. This advanced functionality is implemented on a second Virtex 2000E FPGA using the three types of different units shown in Figure 4.

The controller unit is responsible for coordinating the different units that comprise the ray tracing processor. It queries the list handler units to determine which node of the hierarchy should be traversed next and sends the proper triangles to the intersection unit to accomplish this. The results of the node traversals are then accumulated by the bounding node sorter and passed to the list handler. The list handler can then use this new data to inform the controller units which node to traverse next, and the process continues. Each of these three units is described in the following three sections.

4.3.1. Controller Unit

The controller unit is a complex state machine that is responsible for interfacing with the host computer, and for coordinating all the different units in the ray tracing processor. The unit accepts external request to intersect groups of three rays against the scene currently stored in memory, and returns the nearest triangle that each ray intersects.

Figure 5 is a simplified state model of the controller's function. The first step is to process the root node of the tree and store the result within the list handler unit. Once the node has been processed, the list handler provides an output indicating which node should be traversed next. The next step is to iteratively query the list handler and traverse the nodes suggested. Upon reaching a leaf node,

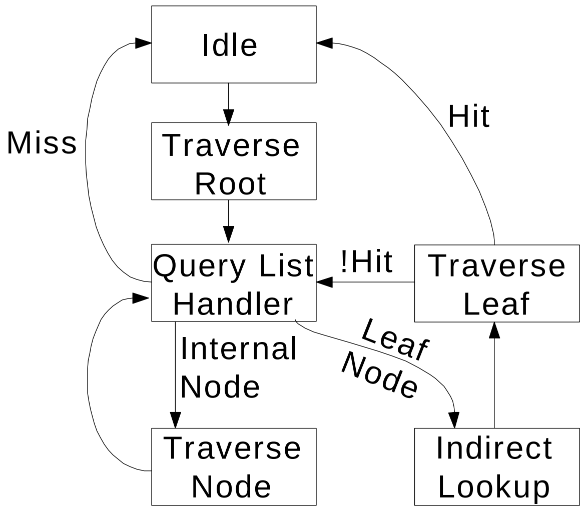

Figure 5. Simplified controller state machine

the controller accesses a table to determine which object triangles should be tested. If the ray strikes an object within the leaf node then the process is complete. Otherwise the controller traverses back up the tree by querying the list handler once more. When the list handler is exhausted, the traversal is complete and the rays are found to not strike any objects.

The control logic requires 360 4 input lookup tables and 367 flip-flops to implement.

4.3.2. List Handler Unit

The list handler unit contains the logic and state storage necessary to perform a nearest to furthest tree traversal. It takes the results of each node traversal as input and outputs which node in the tree should be traversed next. The list handler implements this functionality through the use of three queues.

Each of the three queues represents a different level in the tree. The first queue will contain a sorted list of the intersected children of the root node. The second queue is a list of intersected children from nodes in the first queue, and third queue a list of intersected children from the second queue.

Figure 6 shows a typical traversal operation. First the root node is traversed and the result stored in the first queue, (a). Next the nearest intersected node's children are tested, in this case node 2, and the results written to the second queue, (b). The process is repeated using the nearest intersected node stored in queue two and the result written to queue three, (c). At this point the traversal has reached a leaf node, A. The leaf node A is tested but found not to intersect the ray, (d). Since the leaf node queue, queue three, is empty the next node is queue two is traversed, in this case node 8. Its intersected children are then written to the third queue and the process can continue until an intersected object is found or all three queues become empty.

It is clear from this algorithm that the list handler cannot choose which node to traverse, until the results from the previous traversal have been calculated. Since the ray triangle pipeline is so deep the time necessary to flush the results is very large. To compensate for this problem two list handlers and two control units are instantiated so that they can both share the pipeline. These two sets of control logic process independent sets of rays but share the same pipeline. When one unit is waiting for its results to flush it passes its ownership token to the other control unit so that it can use the data path. This sharing method can completely eliminate pipeline bubbles and make the most efficient use of the datapath.

4.3.3. Bound Object Depth Sorter

To perform a proper nearest to furthest tree traversal it is necessary to have access to a sorted list of intersected child nodes. The process of sorting the traversal results is the job of the bound object depth sorter. This unit takes as input the individual intersect test results for each child node and accumulates them in a sorted list. This list is then transferred to the list handler for later traversal.

The actual implementation of the sorting algorithm is very simple. Each time a new intersect point is found, it is inserted to the sorted list at its correct point. Simultaneously, the displaced nodes are shifted down. This algorithm allows for a one-cycle insertion sort with minimal implementation complexity.

The bounding object depth-sorting unit takes 1159 4 input lookup tables and 617 flip-flops.

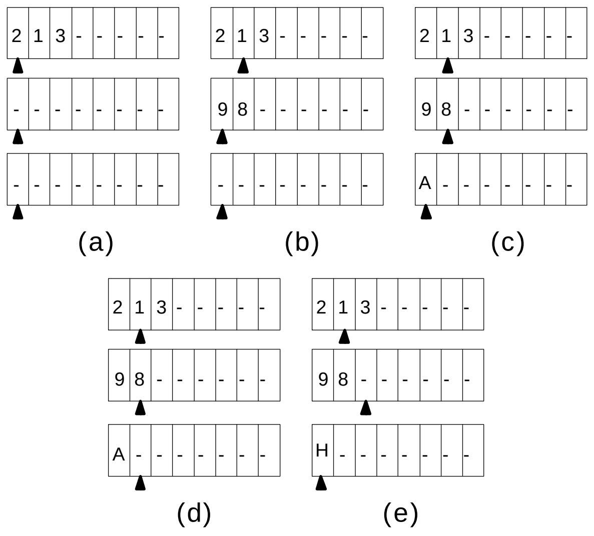

Figure 6. Tree traversal example

Table 1. Prototype ray tracer performance

	Triangles	P4 2GHz Render Time	Prototype Render Time	Performance Increase
Landscape A	51200	6.93s	0.29s	23x
Landscape B	51200	7.03s	0.40s	17.5x
Rain Storm	30848	0.29s	0.81s	0.35x
Shaded Sphere	2048	0.48s	0.17s	2.8x
TM3 Model	1304	0.20s	0.13s	1.5x

5. Prototype Results

In order to evaluate the performance of the ray tracing processor it was necessary to create a simple test jig that could interface with the ray tracer. A simple circuit was created on one of the four FPGAs chips to generate the necessary eye rays and store the results for later retrieval and analysis. This chip also contained a cycle accurate counter to accurately measure performance.

For comparison, the same scenes were rendered on 2GHz Pentium 4 computer using POVray 3.1 [16] with no textures, and the lowest quality setting. This is a freely available ray tracer that is generally accepted to be the best, non-commercial, ray tracer available.

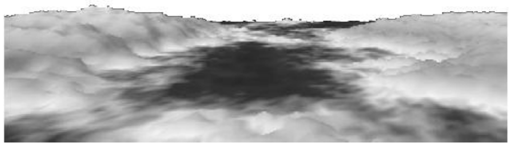

Figure 7. Landscape test image

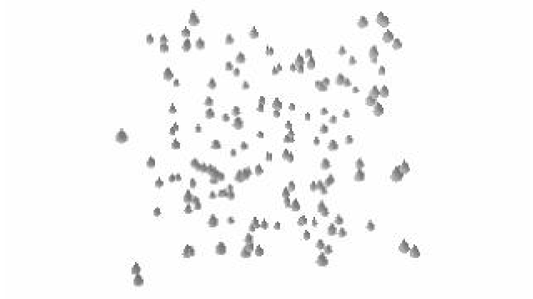

Figure 8. Rainstorm test image

Figure 9. Shaded sphere test image

5.1. Test Scenes

To test the performance of the ray tracing processor a number of different test scenes were used. Figure 7 shows one frame of a landscape animation. This test scene contained over 50,000 triangles in a well-structured bounding hierarchy.

Figure 8 is another high triangle count test image, with 30,848 triangles. The difference between this test set and the landscape is that the random placement of raindrops results in a bounding hierarchy that that does not tightly fit the scene.

Figure 9 is a low triangle count image, with 2048 triangles, and a tight bounding hierarchy. This test set was designed to test performance on simpler scenes.

Also included in the results, but not shown in the figures below, are a 3D model of the prototyping development system, and an additional fractal landscape.

5.2. Test Results

Table 1 summarizes the performance results of the prototype system compared against the POVray 3.1 ray tracer [16]. The two landscape test sets show that the prototype is easily able to out perform software by a factor of 20. This is expected as the prototype system can access memory much faster then a typical desktop computer.

The rainstorm test set yields a different result. The scattered raindrops, combined with their small size, results in a bounding hierarchy that is a very loose fit. This means that the probability of a ray that strikes the bounding volume actually striking a raindrop is very slim. This leads to a large number of leaf nodes being tested only to discover that the ray passes straight through. The software ray tracer does not suffer from this problem, as it can be more adaptive to these degenerative cases. Instead of a fixed hierarchical structure the software approach can use a flat hierarchy of very tight bounding boxes and perform quite well.

The remaining two test sets, the shaded sphere and the TM3 model, are both low polygon count models with relatively tight bounding structures. In these cases the hardware prototype outperforms the software by a more modest margin of a factor of 2. This results from the

Table 2. Projected enhanced system performance

	Triangles	P4 2GHz Render Time	Prototype Render Time	Performance Increase
Landscape A	51200	6.93s	24ms	288x
Landscape B	51200	7.03s	33ms	213x
Rain Storm	30848	0.29s	68ms	4x
Shaded Sphere	2048	0.48s	14ms	34x
TM3 Model	1304	0.20s	11ms	18x

hardware overhead required for each ray no longer being masked by scene complexity as it is with the larger test sets.

For typical large 3D scenes, the hardware ray tracer can achieve significant speedup.

6. Projected Enhanced System Performance

The results presented in the previous section are promising. They show that the simple prototype system that is heavily memory bandwidth-limited can out-perform a state of the art CPU-based system. This section will present a new architecture, based on the prototype system, which is capable of significantly faster performance.

The performance of an FPGA-based ray tracer implemented on a custom development board is dependent on a number of design choices. These choices include: clock speed, FPGA architecture, and memory architecture.

6.1. Clock Speed and FPGA Architecture

The simplest method to increase the performance of a system is to increase the clock rate. In the case of the prototype system the maximum clock rate was limited by memory speed. In the enhanced system this is not an issue as more advanced memory can avoid this problem, as discussed below.

It is also possible to increase speed by adjusting the pipeline depth through the use of automatic pipeline balancing tools. Several trial compilations suggest that a speed of 60MHz could be achieved using this method when targeting a Virtex E series FPGA.

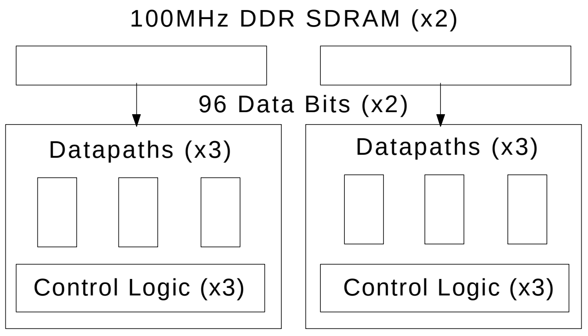

Figure 10. Enhanced System Diagram

To produce even further speed gains it is necessary to switch to a newer FPGA generation with devices with improved architectures and more advanced IC processes. The Virtex II and Virtex II Pro FPGAs [17] include dedicated high-speed multipliers that can be exploited by the ray tracing processor. A single cross product from the datapath can be synthesized to run at over 133Mhz, so it is possible that through fine-tuning the entire datapath could be designed to run at this speed. A safe speed estimate for the datapath would be 100MHz as a synthesis run we performed on the existing datapath to a Virtex II Pro target was able to reach this rate.

An extra bonus of using these dedicated multipliers is a decrease in the number of LUTs required. Instead of utilizing 20,000 LUTs, as in the Virtex E FPGA, the Virtex II allows for an implementation using only 3000 LUTs and 54 dedicated multipliers. This decrease in device utilization will allow for several datapaths to be placed within the same chip. The number of which is ultimately limited by the availability of dedicated multipliers. Using a mid-sized Virtex II Pro chip it is possible to place 4-6 datapaths with ample LUTs remaining for controlling logic.

6.2. Memory Architecture

To utilize a large number of parallel data paths it is necessary to have a memory system that is capable of providing data fast enough to keep the datapaths busy. Since each of the datapaths can consume 184 bits each cycle, equivalent to 2.3GB/s, six independent data paths would require a total of 14GB/s. To avoid this problem a solution similar to that used in the prototype can be implemented. The prototype compared three rays against each triangle to mask the three cycle read time. The same will be done in the enhanced architecture with one minor difference. Instead of processing three rays sequentially through one datapath, three datapaths will each process their own rays. This will reduce the total required memory bandwidth to only 4.6GB/s.

The easiest way to achieve the required bandwidth is through two independent memory banks. Each bank would be 96 bits wide and run at 100MHz and employ the double data rate (DDR) approach. This would result in each set of three datapaths having access to one new triangle every cycle.

Figure 10 shows the complete enhanced system, including two independent sets of three datapath units each fed with a 2.3GB/s memory stream. The bounding hierarchy functionality is implemented through the use of duplicated sets of control logic, and the entire system should run at 100MHz when implemented on a Xilinx XC2VP70.

7. Projected Performance

The enhanced system presented in the previous section implements the exact algorithms that the prototype system does but with twice the clock speed and six parallel datapaths. This results in an increase of performance by a factor of twelve. Table 2 shows the projected performance of the enhanced system on the sample set of test images.

For a large, and well bounded, scene the enhanced system out performs software by over two orders of magnitude. For smaller scenes the performance increase is lower at only one order of magnitude, and for the degenerate case of a poorly bounded scene the performance is only marginally better.

The enhanced architecture also performs well when compared against the current state-of-the-art in hardware ray tracing the AR350 [3]. The Advanced Rendering Technologies product information page for their PURE PCI 3D rendering card states that 8 AR350 processors have a peak ray triangle intersection rate of 1.1 billion per second. This leads to an individual processor rate of 137.5 million intersection tests per second. Our enhanced architecture is capable of 600 million intersection tests per second from a single FPGA, an increase of over four hundred percent.

8. Conclusions

We have implemented a prototype ray tracing system that outperformed a 2GHz Pentium 4 computer by an order of magnitude due to its effective use of memory bandwidth. We project that a larger scale version implemented with newer FPGAs and modern memory could achieve a performance increase of over two orders of magnitudes against software, and a potential increase of four times against an existing hardware implementation.

9. Acknowledgements

The authors would like to thank Xilinx Inc., and MICRONET for research funding, Synplicity for the donation of software tools and David Galloway and Marcus van Ierssel for software and hardware support.

10. References

[1] A. Glassner, *An Introduction to Ray Tracing*, Academic Press, London, 1989.

[2] E. Catmull, *A subdivision algorithm for computer display of curved surfaces*, PhD Thesis, Univ. of Utah, 1974.

[3] D. Hall, *The AR350: Today's Ray Trace Rendering Processor*, In Proc. of the Eurographics/SIGGRAPH Workshop on Graphics Hardware - Hot 3D Session 1, 2001.

[4] J. Purcell, *SHARP Ray Tracing Architecture*, SIGGRAPH Course on Real-Time Ray Tracing, 2001.

[5] J. Schmittler, I. Wald, P. Slusallek, "SaarCOR – A Hardware Architecture for Ray Tracing", *Proc. of the Conference on Graphics Hardware 2002*, 2002, pp. 27-36.

[6] G. Knittel, W. Straber, "VIZARD - Visualization Accelerator for Realtime Display", *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, ACM Press, 1997, pp. 139-146.

[7] H. Pester, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler, "The Volumepro Real-Time Ray-Casting System", *Proc. of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, 1999, pp 251-260.

[8] D. Badouel, "An Efficient Ray-Polygon Intersection", *Graphics Gems*, Academic Press, 1990, pp. 390-393.

[9] J. Stolfi, *Oriented Projective Geometry*, Academic Press, Boston, 1991.

[10] T. Möller, and B. Trumbore, "Fast, Minimum Storage Ray-Triangle Intersection", *The Journal of Graphics Tools*, A. K. Peters, 1997, pp 21-28.

[11] B. Naylor, J. Amanatides, W. Thibault, "Merging BSP Trees Yield Polyhedral Modeling Results", *Proc. of the SIGGRAPH '90 Conference*, 1990, pp 115-124.

[12] H. Samet, *Spatial Data Structures: Quadtree, Octrees, and Other Hierarchical Methods*, Addison Wesley, Reading, 1989.

[13] The Transmogrifier-3a Rapid Prototyping System, http://www.eecg.utoronto.ca/~tm3.

[14] A. Darabiha, J. Rose and W. J. MacLean, "Video-Rate Stereo Depth Measurement on Programmable Hardware", *Proc. of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2003.

[15] A. Alex, J. Rose, and C. Hogue, *Hardware Accelerated Protein Identification*, Master Thesis, Univ. of Toronto, 2003.

[16] Persistence of Vision Ray Tracer, http://www.povray.org

[17] Virtex-II Pro Platform FPGA Handbook, Xilinx Inc, 2002.