Real-Time Raytracer

6.111 Final Project

Adam Lerer, Sam Gross

5/20/2007

Abstract

Raytracing is a 3D graphics algorithm used to generate extremely photorealistic images. This paper describes a highly parallelized implementation of a raytracing algorithm implemented in hardware on an FPGA. The system allows for rendering of planes and spheres, and is able to render shadows and reflections. Objects are shaded using the Phong shading model, which combines ambient, diffuse, and specular shading. Scenes can include simple animation, and users are able to navigate in scenes by translation and rotation.

REAL-TIME RAYTRACER 1

Abstract 1

Overview 3

The Raytracer 4

Design Decisions 5
- Number representation 5
- Vector Normalization & Division 6
- I/O Buses to the Raytracing Units 6

Module Description / Implementation 10
- Master Controller (MC) 10
- Vector Projection 11
- Vector Normalizer 11
- SRAM Controller 12
- VGA Controller 12
- Raytracing Unit (RTU) 12
- Intersector 13
- Divider 15
- Shader 15
- Lights, Shapes, Materials 16

Simulation & Testing 17

Results 18

Future Work 18

Acknowledgements 19

Overview

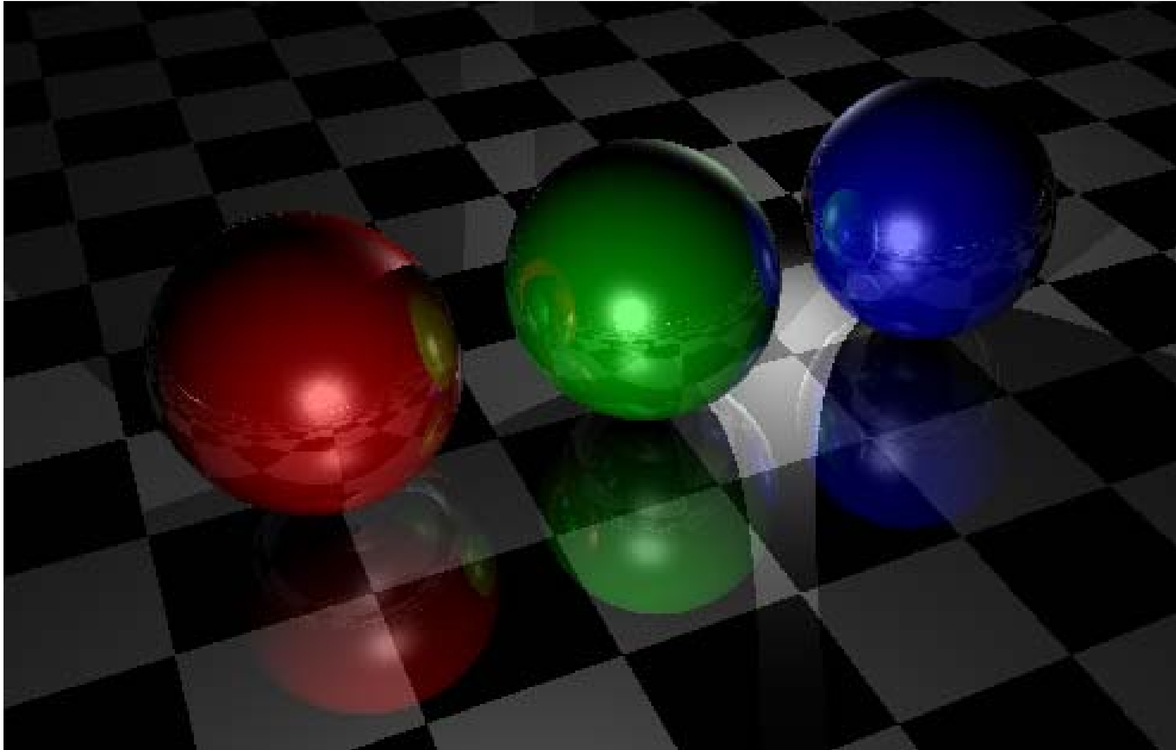

Figure 1: Raytraced Image of Three Spheres on a Plane

Raytracing is a 3D graphics algorithm used to generate extremely photorealistic images. At each pixel of the screen, the raytracing algorithm finds the ray that goes from the viewer's eye through that pixel, and determines which objects in the scene that ray intersects with. A color for the pixel is determined by properties of the closest intersected object and the relationship between the intersection and the lights in the scene. To calculate reflections, the algorithm adds the color of a ray reflected off the intersection point to the color of the pixel.

Raytracing is a very time-consuming algorithm to compute in software, because the calculation of each pixel is independent and calculation of several intersection tests.

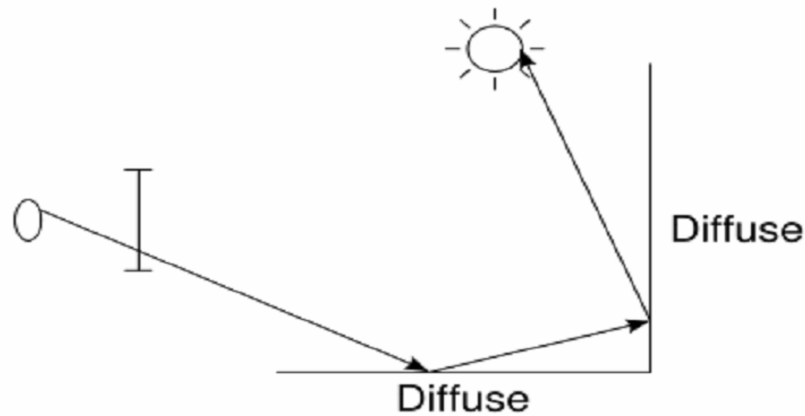

Figure 2: The Principle of Raytracing (Wikipedia)

Each intersection test is also expensive, requiring several multiplications and often expensive operations such as division or square-root. Therefore, real-time raytracing in software is not yet realizable for complex scenes. However, the independence of each pixel allows for easy parallelization, which makes it ideally suited for custom hardware.

The Raytracer

This paper describes a custom digital system that performs raytracing, designed in Verilog and synthesized on the Virtex2-6000 FPGA. The system is made up of several independent raytracing units (RTUs), all controlled by a single master controller. A raytracing unit is given a ray as input and returns the rendered color for that ray. The raytracing unit determines the intersection of the ray with the objects in the scene, and for each light assigns a color to the point based on Phong shading. The raytracing unit also computes reflected rays in order to calculate colors caused by reflection.

The master controller is responsible for calculating initial rays for each pixel on the screen and assigning them to raytracing units. The master controller also synchronizes the different raytracing unit, controls the camera position, and interfaces with the ZBT SRAM controller, which stores the color at each pixel. The VGA controller also

interfaces with the SRAM controller, requesting information to be displayed on the 640x480 VGA display.

Design Decisions

Certain design decisions greatly affected the success of the raytracer.

Number representation

One of the major decisions in the early stages of the project was the representation of numbers in the raytracer. There were three main decisions to be made: (a) whether to use fixed point or floating point numbers, (b) whether to use signed or unsigned numbers, and (c) the bitwidth of each number.

Several considerations needed to be taken into account. First, we would need to use several IP cores including fast multipliers, dividers, and square roots. These IP cores were available for both fixed point and floating point numbers, but although they were somewhat faster and more accurate for floating point, they took a lot more chip area. Since we intended to put as many raytracing units on the chip as possible, this was a major disadvantage. Also, debugging the raytracer with floating point numbers would be much more difficult because it is very time-consuming to interpret floating point numbers from their hexadecimal representation. Therefore, we chose to use a fixed-point representation. The main downside to this approach is that numbers must be between a fixed minimum and maximum; numbers that get too big “wrap around” and become negative. This led to its fair share of ‘overflow’ bugs, but they were pretty easily detectable on simulation and did not cause too many problems.

The choice of bit width was also very important. The most important consideration was that we needed to perform a lot of multiplications of numbers, and the fast

multipliers on the FPGA are 18x18 multipliers; therefore, if our numbers were more than 18 bits, they would require extra multipliers and have extra latency. At first, we were worried that 18 bits would not afford enough accuracy after several operations, but by keeping extra digits after operations like square root and truncating them afterwards, we managed to keep the necessary precision.

We chose to use signed numbers because vectors required signed numbers.

Vector Normalization & Division

Vector normalization and division were very computationally difficult operations to perform on the FPGA. An 18x18 divide took an extraordinary amount of space on the FPGA compared to other operations, and vector normalization would require three multiplications, a square root, and a division in the naïve implementation. For graphics applications, which do not require perfect precision, a far better approach is to perform successive approximations using Newton's method. First, the inputs are bitshifted so that they are between 0.5 and 1. In this interval, a good first approximation can be made using linear interpolation. Then, about two approximations are made using Newton's method, and the final answer is bitshifted back by the appropriate amount. This allowed for a division module with a latency of about 7 clock cycles and a very small LUT count, and a pipelined vector normalizer for the master FSM using 12 multipliers and a throughput of 1.

I/O Buses to the Raytracing Units

One of our main goals in this project was to use as much of the FPGA as possible for parallel computation. One of the problems with using a large percentage of the FPGA is

routing. Luckily, the FPGA would be divided into several completely modular raytracing units, but each of these units needed a 108-bit bus input bus for the initial ray and a 24-bit output bus for the color. We were worried that having many 132-bit buses from the master FSM would cause interconnect problems. Thus, we time multiplexed all the raytracers onto one shared 132-bit bus. This was simple to do with the input bus, because the master FSM simply asserted *start* on at most one raytracer per clock cycle, assigning it the input ray currently on the bus. However, it is possible that more than one raytracing unit could finish in one clock cycle, causing contention on the color bus. Therefore, the raytracing units have an additional input, *req_color*, that asks an RTU that is *done* to assert the color on the shared bus during the next clock cycle. Since the master controller only asserts *req_color* for one RTU per clock cycle, contention is prevented.

Time multiplexing turned out to be very successful, as we were able to fill over 90% of the available slices on the FPGA without interconnect problems.

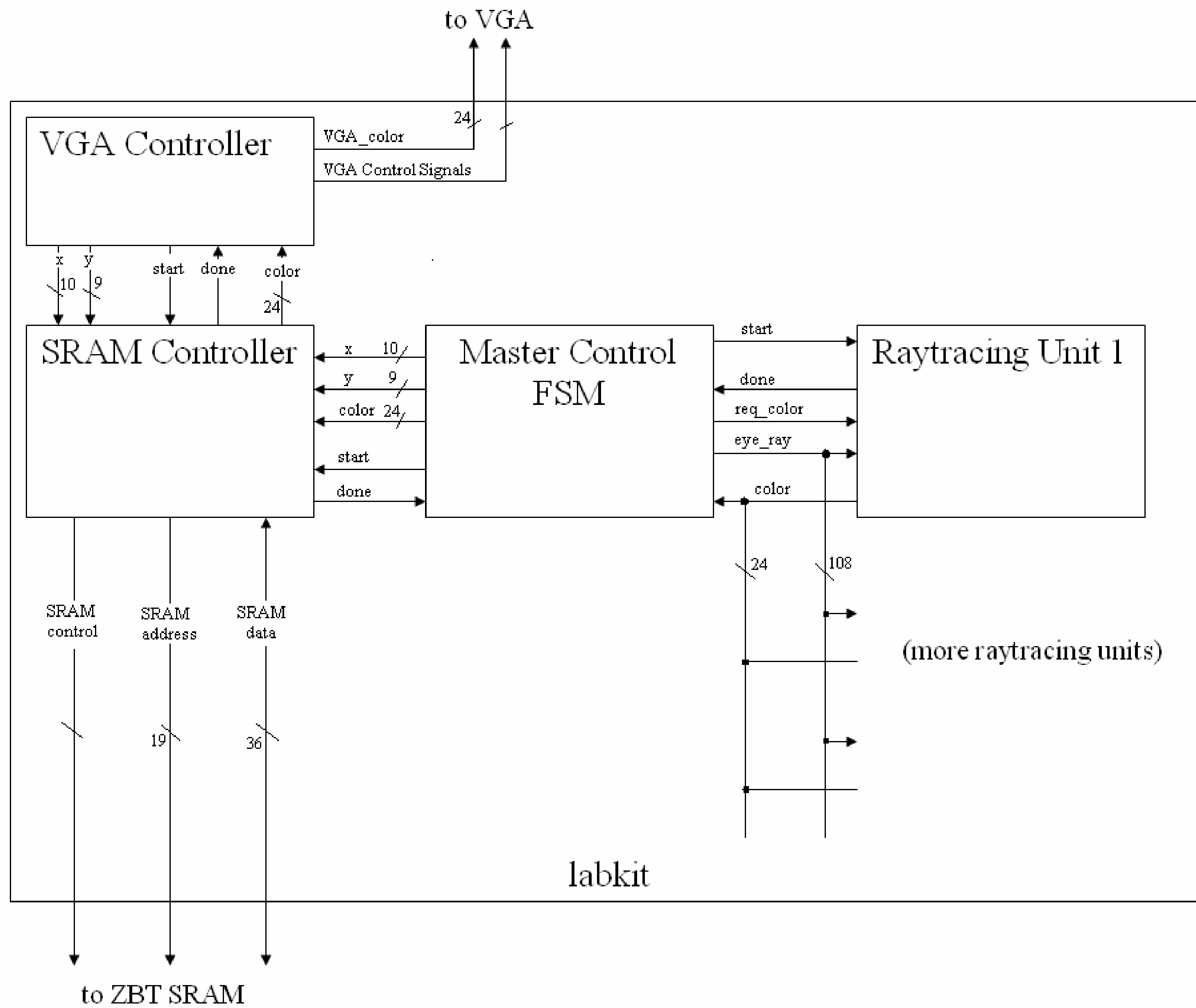

Figure 3: Raytracer Block Diagram

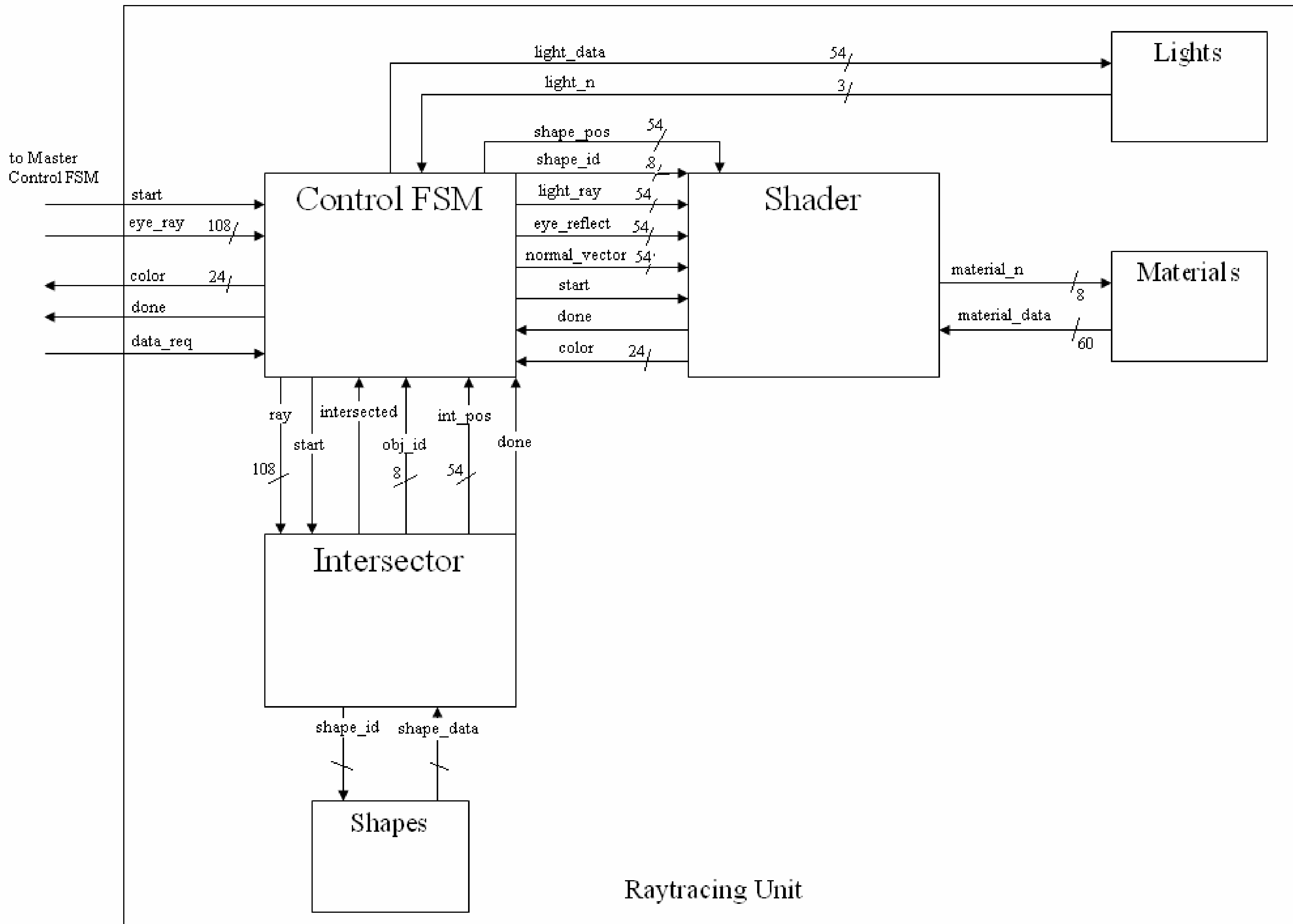

Figure 4: Raytracing Unit Block Diagram

Module Description / Implementation

This section describes the design and implementation of the modules in the raytracers. All modules were designed in Verilog using the Xilinx ISE.

State transition diagrams for the finite state machine (FSM) modules were simplified to be easier to read. In all FSMs, if *reset* is asserted, the FSM transitions to the IDLE state; this takes precedence over all other signals. If a transition is not labeled, it is the default transition.

Master Controller (MC)

The master controller is primarily responsible for delegating work to individual raytracing units and managing their results. The primary input to an RTU is a ray originating at the camera and passing through a pixel on the screen. The master controller has to generate one such ray for each pixel on the screen. In order to generate this ray, the master controller takes the camera data – vectors representing position, direction, and orientation – and passes it to the vector projection module along with the x and y coordinates of a pixel on the screen. The master controller passes the ray generated by the vector projection module to an idle RTU and stores the x and y coordinates of the pixel being processed by the RTU.

The master controller is also responsible for taking the color generated by the RTUs and passing it to the SRAM controller. Color data is passed along a shared bus from the RTUs to the master controller. An RTU only drives the bus when the master controller sets the RTU's *req_color* input high. The master controller sets the *req_color* input high on the lowest numbered RTU that is finished. The master controller passes the x and y coordinates of the pixel and the color data to the SRAM controller.

Vector Projection

The vector projection module is responsible for calculating a normalized vector pointing from the camera to a specified pixel on the screen. The camera's direction and orientation is represented as three vectors: forward, up, and right. The forward vector points in the positive z-direction in the camera's coordinate system. The up vector points in the positive y-direction and the right vector points in the positive x-direction. The projection module scales the right vector by a factor of $c_1 \times (x - WIDTH / 2)$, where c_1 is a constant chosen beforehand to give the desired field of view. Similarly, the up vector is scaled by a factor of $c_2 \times (y - HEIGHT / 2)$. The scaled up and right vectors are added to the forward vector to create a vector pointing in the desired direction. The projection module finally sends the vector through a pipelined vector normalizer.

Vector Normalizer

The vector normalizer module uses Newton's method to compute an accurate approximation of the reciprocal of the square root vector magnitude. After the magnitude is computed, normalizer bit-shifts the sum of squares left so that it is between 0.5 and 1. A linear approximation is used for the initial guess, since the function 1/sqrt(x) is nearly linear between 0.5 and 1. Newton's method gives us the approximation $x_n = x_{n-1} \times (1.5 - 0.5 \times k \times x_{n-1}^2)$ where x is the current guess and *k* is the original number. We do two iterations using Newton's method to get successively better approximations. If we shifted the original number by *j*, we shift the answer left by floor(*j*/2) and then multiply by sqrt(2) if j is odd. Altogether, the fully pipelined version of the normalizer has a latency of 10 clock cycles and uses thirteen multipliers. The un-pipelined version has a latency of sixteen clock cycles and uses one multiplier.

SRAM Controller

The SRAM controller is responsible for interfacing with the ZBT SRAM on the labkit and implementing a double buffer using page-flipping. The SRAM controller writes to one SRAM while reading from the other. When the flip input is asserted, the controller switches buffers, reading from the SRAM to which it was previously writing and vice versa. Since the data signal to the ZBT SRAM should be delayed two clock cycles relative to the address, the controller passes the input data through two registers before sending it to the SRAM.

VGA Controller

The VGA controller is nearly the same as the VGA controller used in Lab 4. The only significant difference is that VGA controller used in the raytracer interfaces with the SRAM controller instead of a hard-coded display field.

Raytracing Unit (RTU)

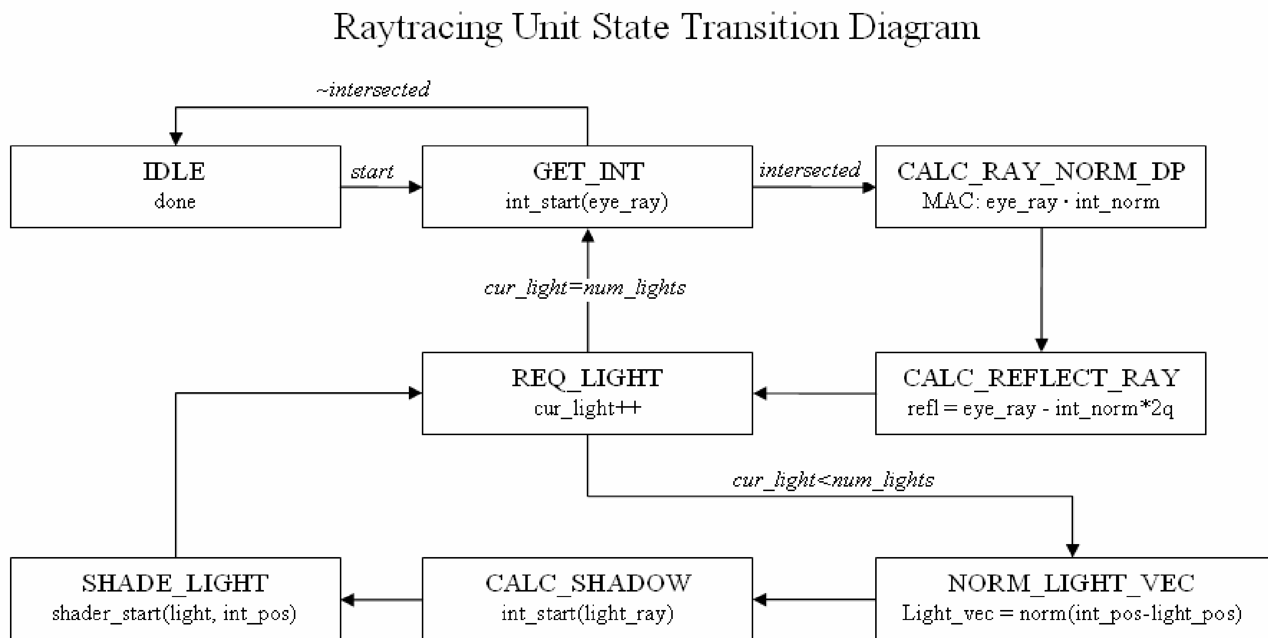

Figure 5: State Transition Diagram for the Raytracing Unit

A raytracing unit is the main processing engine of the raytracer. The complete raytracing unit consists of a control FSM, intersector and shader modules, and modules that contain the lights, shape, and material data. The control FSM is responsible for delegating to and controlling the intersector and shader modules, as well as computing the reflected ray from the camera off an object. When the start input is asserted, the controller initializes by setting the current color to black, the current ray to the passed eye ray, and the current depth to zero. The controller next delegates the task of finding an intersection of an object in the scene with the current ray to the intersector module. If no intersection is found, the process is finished – the current pixel will be colored black, which is the background color. If an intersection is found by the intersector module, the control FSM next computes the reflected eye ray by adding the normal scaled by a factor of 2×(normal · current ray) to the current ray.

After the reflected ray is computed, the control FSM loops through each light in the scene. The control FSM first checks if an object blocks the path from the light to the point of intersection. The control FSM does this by constructing a vector from the point of intersection to the light. This vector is then normalized by an un-pipelined version of the vector normalizer. The control FSM uses the intersector to check if there is an object between the light and the point of intersection. If there exists such an object, the original object is shadowed and the contribution from the light is not calculated. Otherwise, the control FSM uses the shader to compute the lighting contribution of the specified light. The lighting contribution is added to the current color. In order to compute reflections, this process is repeated with the depth incremented and the current ray set to the reflected ray until the depth reaches five or no intersection is found. Lighting contributions from subsequent reflections contribute by a factor of 2^{-depth} which simulates a reflectivity of one-half.

Intersector

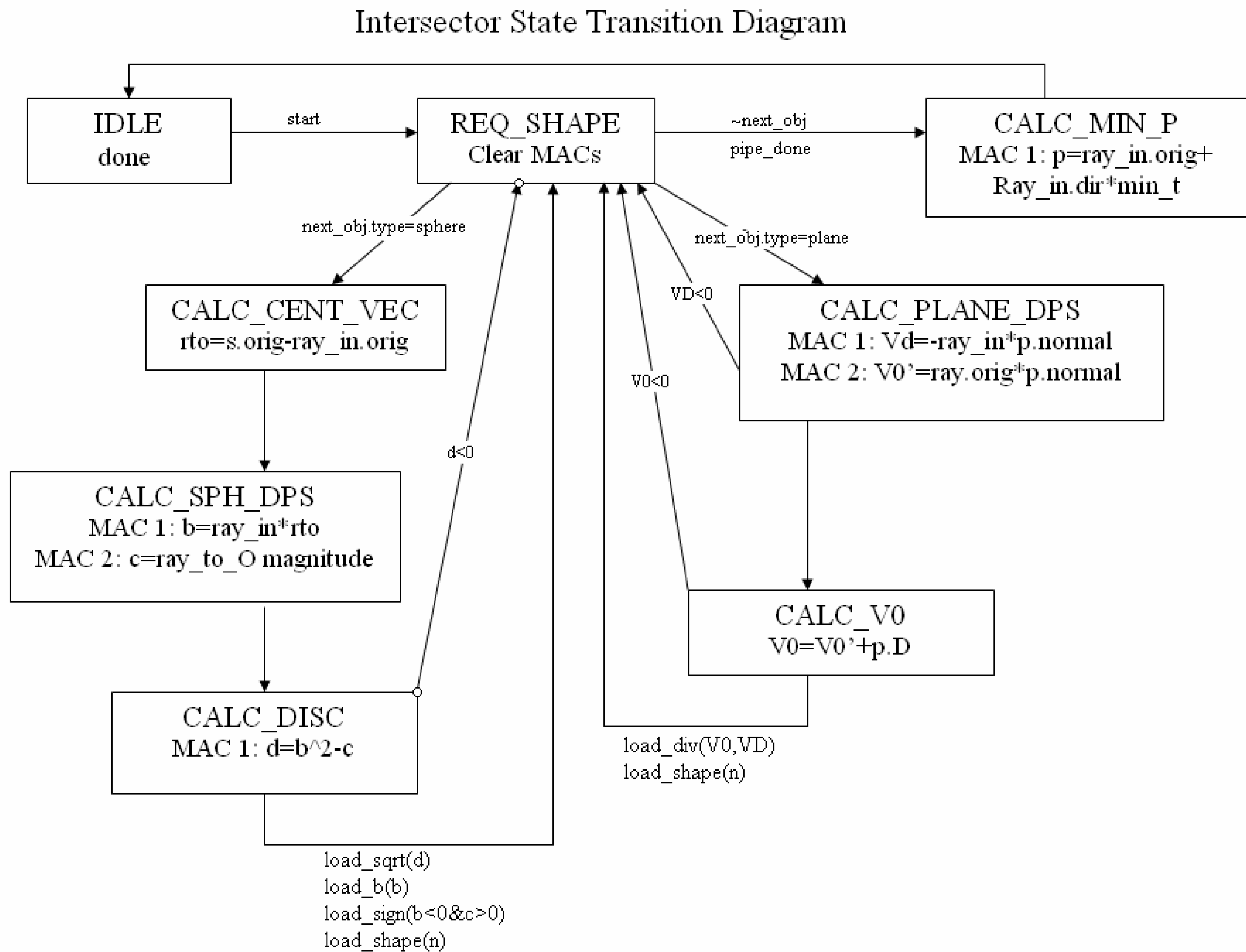

Figure 6: State Transition Diagram for the Intersector

The intersector modules is responsible for computing whether the given ray intersects an object in the scene and the point of intersection and object's normal at that point for the closest intersection any occur. The intersector contains within it the logic for calculating intersections of rays with both spheres and planes, the two shapes supported by the raytracer. Although the shape and plane logic are mostly independent, both are contained within the intersector in order to use the same multiply-accumulators for both sphere and plane intersection tests.

Sphere and plane intersection tests require a series of multiplications and additions as well as one "expensive" operation each. Since sphere intersection requires solving the quadratic equation,

computing a square-root is necessary. The pipelined CORDIC coregen module is used to compute the square-root. Plane intersection tests requires computing a division, which is done in the divider sub-module.

Both the plane and sphere intersection tests compute a distance, *t*, along the eye vector that an intersection occurs. To get the point of intersection we simply scale the eye vector by a factor of *t*. To compute the normal at the point of intersection on a sphere we subtract the origin of the sphere from the point of intersection. We then normalize this vector by multiplying it by a factor of one over the radius of the sphere – a pre-computed value stored in the shapes module. The normal of a plane is the same at all points and is part of the data describing the plane.

Divider

We use Newton's method to approximate the reciprocal of the denominator. We first normalize the denominator by shifting it left until it is between 0.5 and 1, much like we do when normalizing vectors. We use two iterations of Newton's method to compute the reciprocal of the denominator. We then multiply the reciprocal of the denominator by the numerator and shift left again by the same amount to obtain the quotient. Since all numbers are required to be between 0.5 and 1 we only perform a division operation if the numerator is less than the denominator.

Shader

The shader module computes the diffuse and specular contributions of a light. Diffuse shading is computed by multiplying the object's color by a factor of the dot product of the object's normal and the vector to the light. This is based on the fact that objects facing towards a light are brighter than

objects angled away from a light. The specular component simulates the reflection of the light into the

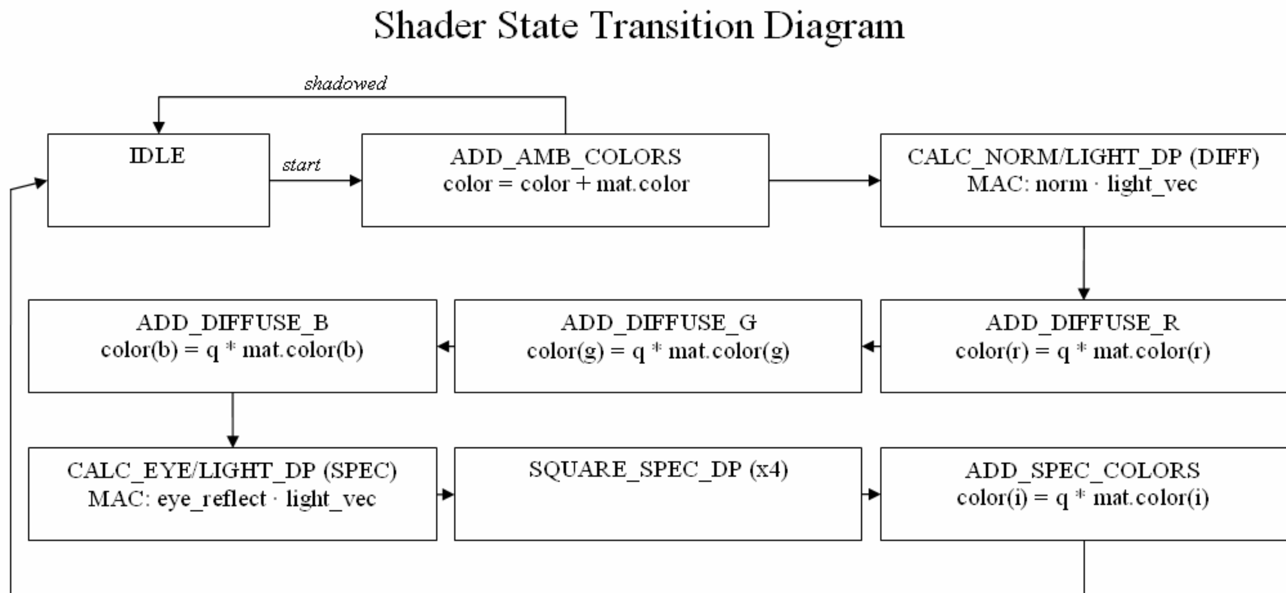

Figure 7: State Transition Diagram for the Shader

viewer's eye on shiny objects. We compute the specular component by taking the dot product of the reflected eye vector and the vector from the object to the light. We then raise this value to the sixteenth power to get a more focused bright spot. For example when the reflected eye vector is incident with the vector to the light the dot product is one, since light is being reflected directly off the object into the viewer's eye. We sum the specular and diffuse component to get the contribution from the specified light.

Lights, Shapes, Materials

The lights, shapes, and materials modules describe read-only memories (ROMs) containing the positions of the lights and shapes and the geometry, color, and reflective properties of the shapes. The modules are described in such a way so that they can be synthesized either to LUTs or block RAMs depending on which resource the synthesizer decides is more available.

Simulation & Testing

Our first step, even before the project proposal, was to write a proof-of-concept software program in Java that rendered the scene consisting of the three spheres and a plane. We used this experience to decide what we could feasibly implement on an FPGA in the allotted time. The Java program also proved immensely useful in testing and debugging, once we started using the same fixed-point arithmetic in the Java program as we were in Verilog. We used ModelSim to run behavioral simulations of the Verilog code. Since we knew that the Java program produced the desired result, we were able to compare values from the Java program to the results from behavioral simulations. For example, we often checked if the intersector module calculated exactly the same point of intersection and normal vector as the Java program for a given input ray.

Our most common source of error was integer overflows due to the use of fixed-point numbers. Often, this caused visual artifacts, such as "bullet" holes appearing in spheres. Different fixed-point operations required separate methods to detect overflows. Often we detected if the result of the operation had a sign different than what was expected. For example, adding two positive numbers should always yield a positive number, while adding two negative numbers always yields a negative number. (Adding a positive number to a negative number never overflows.) When we detect an overflow, we set the number to either the largest or smallest fixed-point number, depending on whether the overflow is positive or negative. We intentionally decided to limit our range to between −1 and 1 so that multiplication could not cause overflows.

As we increased the number of RTUs, we encountered timing issues on the shared output data bus from the RTUs to the master controller. This was caused by increased latency between the master controller asserting *req_color* and the RTU asserting the color on the output bus. To solve this problem, we registered *req_color* in the RTUs and asserted the color a clock cycle later. Since we

were giving the RTU a new ray to work on during the same cycle it finishes, but only retrieving the color from the RTU one cycle later, some buffering was required to accomplish this.

Results

The raytracer implementation met all of the design specifications and implemented almost all of the features that were proposed in the design phase. Planes and spheres were both rendered and shaded with ambient, diffuse, and specular shading under multiple lights. Reflections and shadows were both rendered successfully. A small camera module allowed for zooming, panning, and rotating the view being rendered, and simple animation of the scene was accomplished. A scene with three spheres and a checkerboard plane were rendered at a frame rate of between 8 and 20 frames per second, depending on the particular camera placement and the complexity of the area within the camera view. This was accomplished by synthesizing 14 RTUs on the Virtex2-4000 running in parallel, which consumed over 90% of the available slices on the FPGA.

Future Work

There are several directions in which this project could be taken in the future.

1. Modular scene information: In the current implementation, all scene information is stored internally in the individual raytracing units. This allows for very efficient access of the scene data, but it makes it very difficult to modify scenes efficiently after synthesis. In a more developed version of this project, the scene information would be stored in block ram and accessible through an interface module by the individual raytracers. Then, scene information could be loaded onto the block rams after synthesis.

2. Shapes: For this project, we only implemented planes and spheres. However, most complex scenes are made up of other shapes, triangles in particular. Future implementations should allow

triangles to be rendered, because more complex shapes could then be rendered as amalgamations of triangles.

3. Multiple FPGAs: Since only ray and color information needs to be transferred to the different raytracing units, and since the architecture of the raytracer is so parallelizable, the design could be extended to run on multiple FPGAs. There would be one 'master' FPGA and a number of 'slave' FPGAs made up of several raytracing units and a communication module that reported pixel colors to the master FPGA. Theoretically, a linear speed increase in the number of FPGAs could be realized.

4. Scene data structures: The main algorithmic problem with this implementation is that the time to find an intersection grows linearly with the number of objects in the scene. This is not at all practical for scenes with hundreds or thousands of objects. Software implementations of raytracers run in time logarithmic in the number of objects, because they preprocess the objects into data structures, keeping track of which objects occupy which part of the scene so they don't have to be checked for each ray. This would be quite complex to implement on an FPGA, because they are dynamic-memory data structures, so it might be more effective to implement a small processor on the FPGA that handles the data structures (scene-graphs / KD-trees) and interfaces with the various intersector modules, giving them the appropriate objects to check for a given ray.

Acknowledgements

We would like to thank:

Javier Castro for being a great TA and working with us on all stages of the project.

Gim Hom and David Wentzloff for their help and advice.

Anantha Chandrakasan for his support, and for warning us not to do MPEG encoding.

Nathan Ickes for providing us with a module to reduce clock skew to the ZBT SRAMs.

Ben Gelb for convincing us to take the class and his encouragement.

Igor Ginzburg for his wise words, “If you can’t write it in Java in a night, don’t do it.”