NTNU
Det skapende universitet

Modellering av en hardware ray tracer

Erlend Brataas

Master i elektronikk
Oppgaven levert: Juni 2011
Hovedveileder: Bjørn B. Larsen, IET

Norges teknisk-naturvitenskapelige universitet

NTNU – Trondheim
Norwegian University of
Science and Technology

Department of Electronics and Telecommunications

Master's thesis

Design of a hardware ray tracer

Author:
Erlend Brataas

Supervisors:
Dr. Øystein Gjermundnes
Associate Professor Bjørn B. Larsen

June 27, 2011

Contents

List of Figures

1 Abbreviations

ALU Arithmetic logic unit.

FPGA Field-programmable gate array.

FPS Frames per second, or frame rate.

HDL Hardware description language.

RGB Red, green, and blue (color model).

RTL Register transfer level.

2 Introduction

A ray tracer is a technique for generating an image by tracing the path of light from a camera's point of view through pixels in an image plane and simulating the effects of its encounters with objects. The technique is capable of producing a very high degree of visual realism since e.g., light reflections and other optical effects are simulated. The downside of ray tracers are that this technique can be very compute intensive. This technique is poorly suited for real-time applications. It is very well suited for applications where the image can be rendered slowly ahead of time.

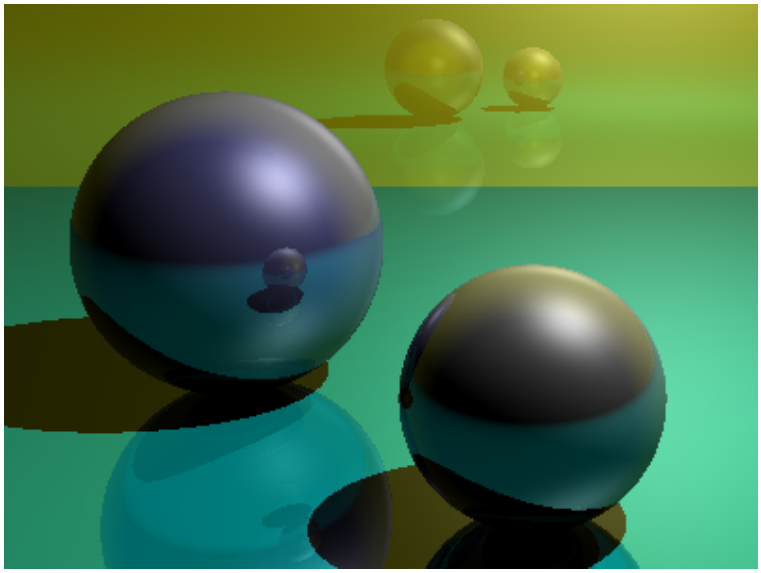


Figure 1: Image generated by a ray tracer.

2.1 My contribution

There already exists numerous ray tracers written in C++. There are fewer ray tracers written in hardware description languages (HDL). Many of these HDL ray tracers are complex and have plenty of functionality. My contribution to this topic is to create a simple customizable ray tracer with a framework around it in form of tools written in C++ and Verilog that simplifies the design process, and automates the verification process. The ray tracer is simple in the sense that only some few optical effects are implemented, but the framework of the ray tracer is more complex where it support other optical effects. This ray tracers and its framework in form of tools and its fundamental functionality, can therefore be used by other as an introduction to ray tracers written in HDL languages.

3 Modeling of the Ray Tracer in C++

3.1 3D scene

The 3D scene that is to be visualized through a 2D plane, is built up of geometric primitives of type sphere and plane. Geometric primitives are the simplest atomic geometric objects that the ray tracer can handle. There are several other ways to build a scene. Rather than using spheres and planes, one could use polygons or points. Ray tracing with geometric primitives like spheres and planes is a good starting point for beginners within ray tracing, since it is mathematically easy, and intuitive to compute how light behaves when light intersects with spheres and planes, that is if the model of the light is simplified as in section 3.4. A sphere can be modeled using only the spheres center and radius. A plane can be modeled using only the planes normal vector and distance from origo.

The 3D scene is modeled using a Cartesian coordinate system with x, y and z as the coordinate axis. Any point in the Cartesian coordinate system can be represented as a vector from origo.

$$\vec{v} =< v_x, v_y, v_z > \tag{1}$$

3.2 Ray

A light beam, or a ray, is modeled as a vector. Every ray $\vec{r}$ has an origin $\vec{o}$, direction $\vec{d}$ ($\vec{d}$ is a unit vector) and length t (t is a scalar).

$$\vec{r} = \vec{o} + t\vec{d} \tag{2}$$

3.3 Calculation of Sphere and Plane intersection

In this section we will see some mathematical expressions that require a relatively large portion of the total execution time, namely checking if a ray intersects with a geometric primitive. Later in section 3.4 we will see how light behaves if there is an intersection. The following equations are common knowledge, equation 8 is derived by my self.

3.3.1 Sphere intersection

Assume that a ray is starting from point $\vec{o}$ with direction $\vec{d}$, as in equation 2. A sphere is located with its center at $\vec{c}$ and has a radius r. The distance t between $\vec{o}$ and the surface of the sphere, in direction $\vec{d}$, is as shown in equation 4.

$$\vec{v} = \vec{o} - \vec{c} \tag{3}$$

$$t = -(\vec{v} \cdot \vec{d}) \pm \sqrt{(\vec{v} \cdot \vec{d})^2 - (\vec{v}^2 - r^2)} \tag{4}$$

If the part inside the root of equation 4 is positive or zero, and all solutions of t are positive or zero, the ray intersects with the sphere and the ray origin is not placed inside the sphere. Then if t has two equal solutions, the ray just touches the sphere's surface, if t has two different solutions, the ray has a path through the sphere.

3.3.2 Plane intersection

Assume that a plane has a normal unit vector $\vec{n}$, and $\vec{q}$ is a point in the plane. $\vec{r}$ is then a point on the plane if

$$\vec{n} \cdot (\vec{q} - \vec{r}) = \vec{0} \tag{5}$$

If $\vec{r}$ is a ray, equation 2 can be inserted into equation 5, we then get

$$\vec{n} \cdot (\vec{q} - \vec{o} - t\vec{d}) = \vec{0} \tag{6}$$

Equation 6 can be rearranged to

$$t = \frac{\vec{n} \cdot \vec{q} - \vec{n} \cdot \vec{o}}{\vec{n} \cdot \vec{d}} \tag{7}$$

$\vec{q}$ is still a point in the plane if $\vec{q}$ is set to $q \cdot \vec{n}$, where q is the distance from origo and the plane in direction $\vec{n}$, where $\vec{n}$ is the plane normal vector. If $\vec{q} = q \cdot \vec{n}$ is inserted into equation 7 we get

$$t = \frac{q - \vec{n} \cdot \vec{o}}{\vec{n} \cdot \vec{d}} \tag{8}$$

The ray is parallel to the plane if $\vec{n} \cdot \vec{d} = \vec{0}$. If $\vec{n} \cdot \vec{d} \neq \vec{0}$ and $t < 0$, the plane is behind the ray. Therefore, if $\vec{n} \cdot \vec{d} \neq \vec{0}$ and $t > 0$ there is an intersection between the ray and the plane with a distance of t from $\vec{o}$ in direction $\vec{d}$. How testing for intersection with planes is actually done in the ray tracer model, is shown in section 3.10.2.

3.4 Behavior of Light

Simulation of light passing through a scene, is a complex and compute intensive task. The modeling of light is therefore simplified. In this simplified model, four different phenomena can happen when light hits an object, depending on the properties of the object.

Reflection Light can be reflected on an object.

Refraction Light can pass through an object with change of angle.

Absorption Light can be absorbed by an object.

Pass Through Light can pass through an object with no effect.

Reflection and refraction are modeled using reflection and refraction rays. Light passing through objects with no effect has not been taken into account in this model. Most objects do not have completely smooth surfaces, all objects has to some degree an irregular surface where reflected light is scattered in almost all directions. Scattered light is modeled using shadow rays. Shadow rays are also used to model direct light reflected by objects with smooth surfaces. Absorption of light is embedded in how light propagate through all these rays. It is thus three types of rays in this simplified model. A more detailed description of these rays are found later in this thesis.

Open air rays are reflection rays, rays that just have exited the object it was inside, and other open air rays that have not been reflected or refracted yet. In this thesis, only reflection rays and rays coming from the eye are relevant open air rays.

Refraction rays are rays that exist inside objects.

Shadow rays are rays spawned from an intersection point to a light sources. These shadow rays accumulate illumination from the light source.

In the real world, photons emitted from a light bulb are casted in almost all directions. Some of these photons may hit the eye after they have interacted with the environment, but most of them will not. To generate an image with sufficient quality by using this technique on a computer, would require immensely long computation time. Therefore this process is reversed where the rays are shot from an eye through the image plane, and then farther on, what happens next, is described in the following sections.

In the picture below, one can see the eye, image plane, light source, some objects, open air rays (red), refraction rays (green) and shadow rays (yellow). Open air rays are reflection

rays, rays spawned from the eye, and rays that exit objects. These rays are also described more carefully in the following sections. The image plane and the eye is described in section 3.5.

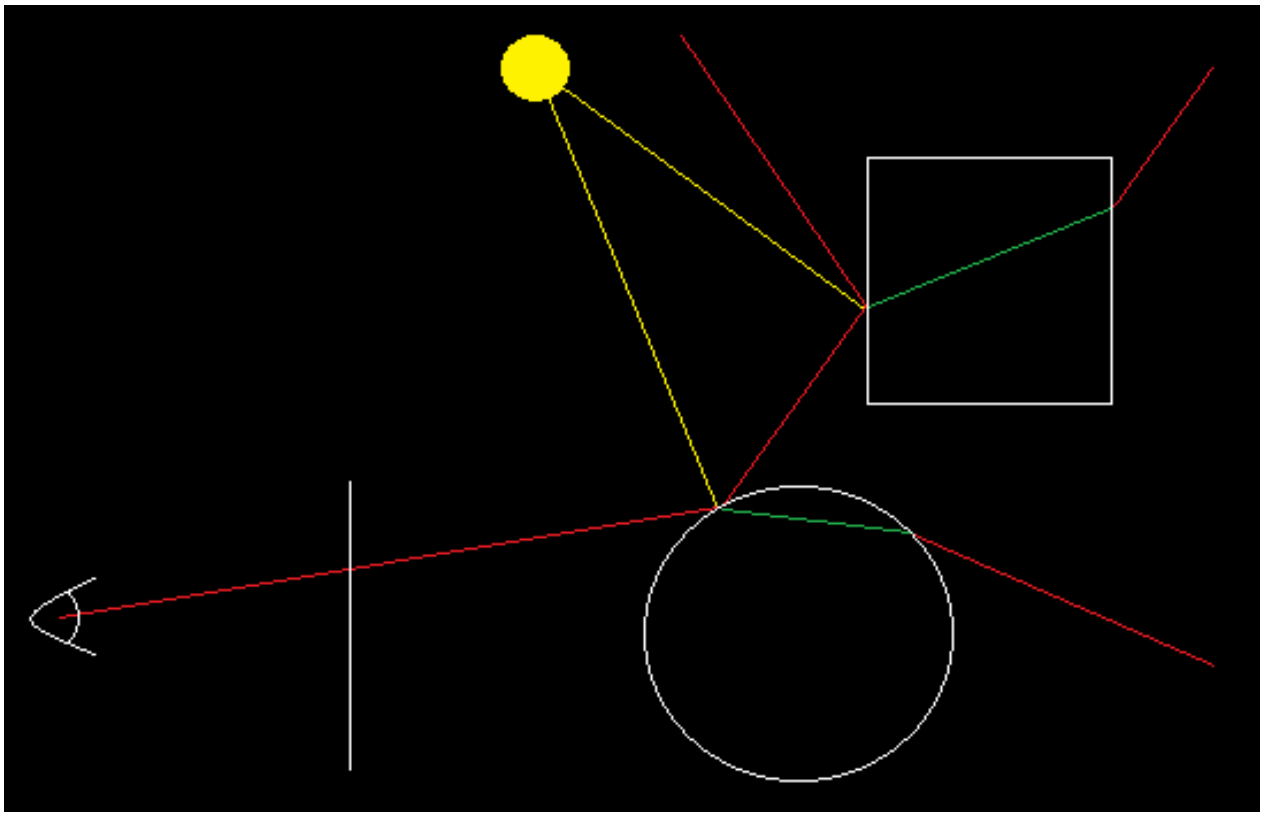


Figure 2: Open air rays are colored red, shadow rays are colored yellow, and rays inside objects are colored green.

3.4.1 Reflection and Refraction

Every ray that hits the surface of an object can spawn a new reflected ray if the object is able to reflect. The amount of light reflected is specified by the objects reflection property. The point where the ray intersected with the object, becomes the origin of the new reflected ray. The direction of the new reflected ray $\vec{d_{refl}}$ is as given in equation 9 where $\vec{n}$ is the surface normal vector and $\vec{d}$ is the direction of the ray to be reflected.

$$\vec{d_{refl}} = \vec{d} - 2 \cdot (\vec{d} \cdot \vec{n}) \cdot \vec{n} \tag{9}$$

Refraction rays are not used in this thesis. The reason for not using refraction rays are that the time frame for completing this thesis is 20 weeks, to avoid getting to many delays

and not finishing the project, some functionality had to be excluded in this paper, but the framework in section 3.6 that supports this functionality is implemented. This framework allows refraction, and other rays to be implemented in the future. Other technologies that may be implemented later are described in section 7.1.

3.4.2 Shadow rays and computation of diffuse and specular light

Diffuse light is light that is reflected by an irregular surface where the reflected light is scattered in almost all directions. Specular light is light that is reflected by a smooth surface. These two phenomena are only used in the context of where the light is coming directly from a light source, not where the light is reflected between objects themselves. These two sources of light can be added together to get a more realistic illumination. These two light sources make up the shadow (yellow) rays that are shown in figure 2.

How light coming directly from a light source will affect objects is specified by the objects diffuse and specular property. The reason for using different parameters on reflection between objects, and reflection between objects and light sources is because it is of interest to specify how much light that is coming directly from a light source will affect different objects, and how much light is reflected between different objects themselves. The amount of diffuse light reflected by the surface is calculated using the dot product between the surface normal $\vec{n}$ and the unit vector from the intersection point to the light source $\vec{l}$, as shown in equation 10.

$$diffuseIllumination = diffuseProperty \cdot (\vec{n} \cdot \vec{l}) \tag{10}$$

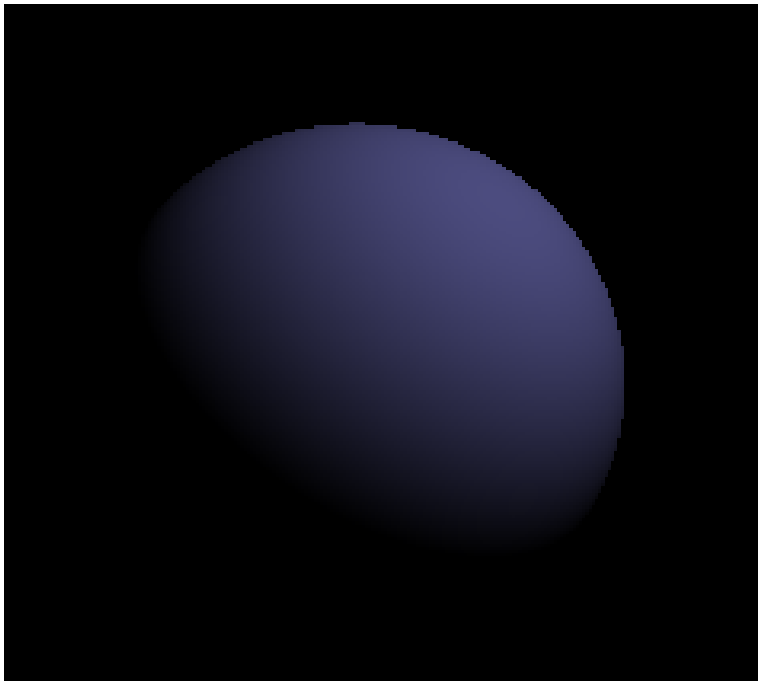


Figure 3: Diffuse reflection

Another source of light is specular reflection. This is reflection created by a smooth surface. If a reflection ray finally hits a light source, a bright spot will be visible on the surface. Computation of the specular reflection follows this equation

$$specularIllumination = specularProperty \cdot (\vec{d} \cdot \vec{l})^{20} \tag{11}$$

This light spot will move around on the object when the camera (or the eye) moves around. Total illumination coming from the light source is the sum of these illuminations, that is

$$illumination = diffuseIllumination + specularIllumination \tag{12}$$

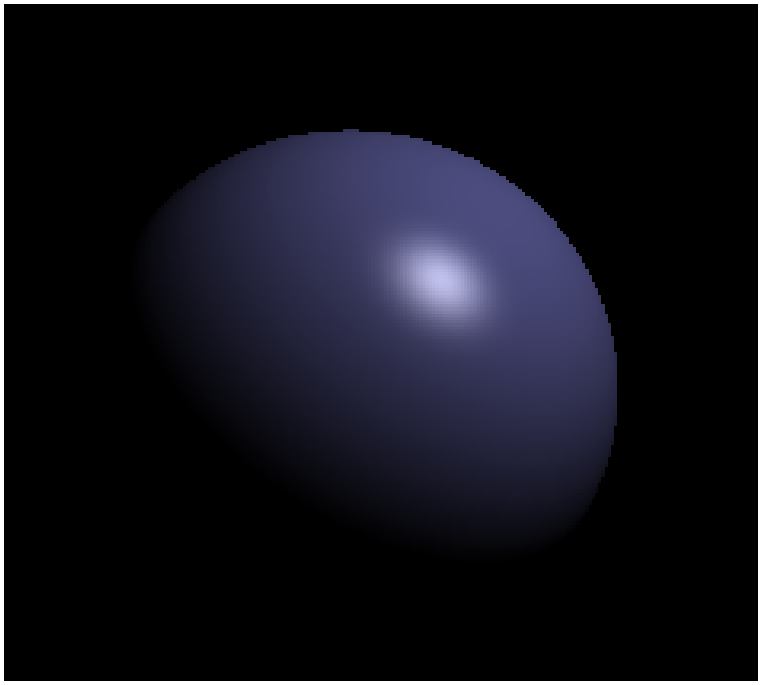


Figure 4: Diffuse and specular reflection

Specular illumination only care about the color of the light source, where diffuse illumination also take into account the color of the geometric primitive.

3.5 Image Plane and Camera Management

The 3D model scene is projected onto a 2D image plane. In this project, the image plane is 800 pixels wide and 600 pixels high. To determine the color of each pixel in the image plane, a ray is shot through each pixel in the image plane from an eye. These rays may hit objects along their path.

Since the image plane has fixed with and height, the distance between the eye and the image plane determines what wide-angle lens one get for the eye (or camera). The closer the image plane is to the eye, the wider wide-angle one get. Increasing the wide-angle gives also a perception of zooming out. It does not matter if the image plane is placed on the opposite side of the scene since the purpose of the image plane is only to set the direction of the 800 times 600 rays that are shot from the eye. This image plane therefore insures a perfect 3D to 2D projection of the scene that can be displayed onto a computer screen or other displays.

The position of the eye is the origin of the rays that are shot through the image plane. This

is the position that the scene is viewed from.

Initially the eye is looking horizontally straight ahead, where the image planes center has the same height as the eye and the image plane is aligned to all three axis in the 3D scene. To tilt, rotate or pane the eye (or camera), each ray that is shot through the image plane can rotate around all three axis. Since a direction is actually a vector with dimension of 3, a rotational matrix is used to rotate the direction of the rays shot from the eye through the image plane. For simplicity's sake, rotation around one axis, only the x axis will be discussed. The rotational matrix around the x axis is as shown in equation 13

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{13}$$

$$d = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{14}$$

$$d_{rot} = R_x(\theta)d \tag{15}$$

Where θ is the desired angle one want to rotate the vector around the x-aksis. Equation 14 shows the direction on vector form, and equation 15 shows how to compute the new direction vector. Equation 15 can be presented on the following form

$$\begin{aligned} x_{rot} &= x \\ y_{rot} &= y \cdot cos(\theta) - z \cdot sin(\theta) \\ z_{rot} &= y \cdot sin(\theta) + z \cdot cos(\theta) \end{aligned} \tag{16}$$

An image of how the eye, image plane and scene relate to each other is shown in figure 5.

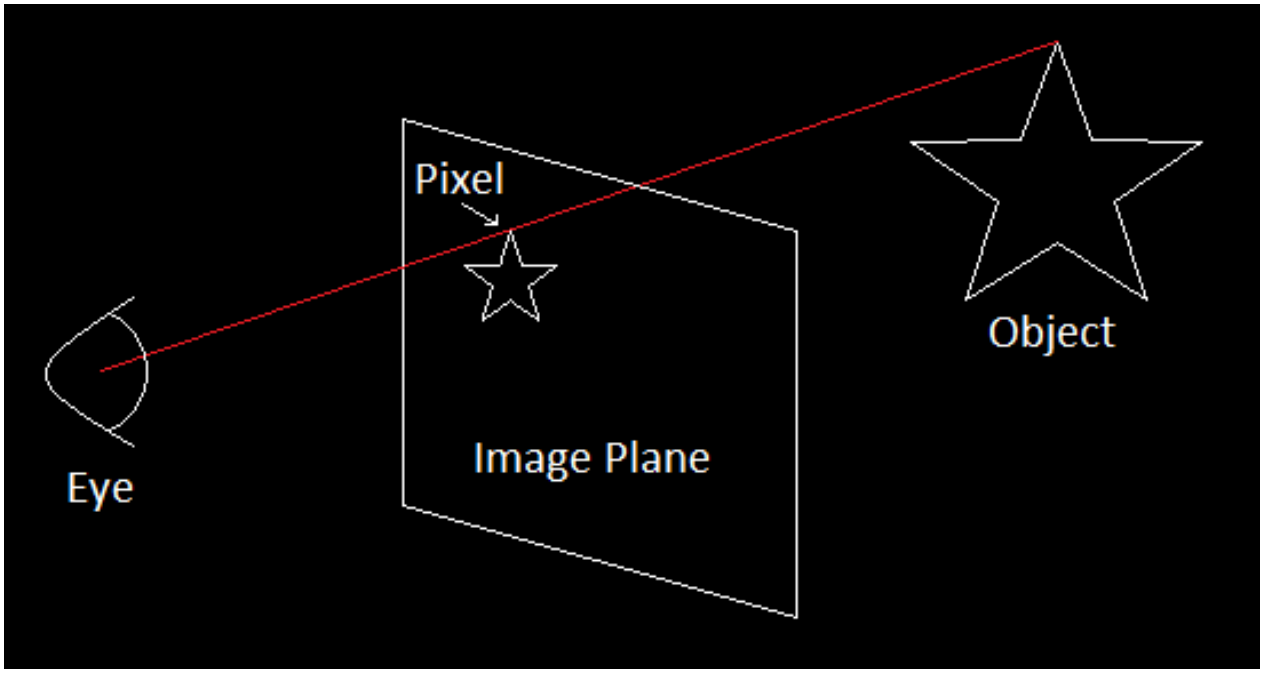


Figure 5: Scene

3.6 Tracing a ray, and collecting information

As described in section 3.5, rays are shot through each pixel in the image plane. The color of each of these pixels can be computed independently and parallel to all other pixel colors. Therefore we will now study how color and other information flows through all rays derived from one single ray, spawned from the eye through one single pixel. We will also study how all these rays relate to each other.

3.6.1 Information carried by open air rays

As described earlier, each ray has information about its origin and direction. Each open air ray also carries color, and reflection information.

The Color information is of RGB type. Each red, green and blue color component, range from 0 to 1, where 1 implies maximum presence of a color and 0 implies no presence of a color.

Reflection value determines how much diffuse and specular light, coming from a light source, is reflected indirectly via other objects. Light from light sources hitting objects directly is

specified by the objects diffuse and specular value.

How all this information, and other information is accumulated, and transformed into accurate color information displayed on the screen, is discussed later on, but first we need to understand how the ray structure is built, and how one iterate over this structure.

3.6.2 Ray structure

If an open air ray (red) intersect with an object, and a light source is visible from this intersection point, illumination is added to the rays color, this ray can spawn other rays like reflection and refraction rays. A reflection ray, and a refraction ray finally exiting the object it was inside, can spawn other rays if they intersect with other objects that can reflect or refract. All these rays form a tree structure with the root ray coming from the eye. A model of this structure with open air rays, that is, ray coming from eye, reflection rays, and refraction rays finally exiting the object it was inside, are shown as red rays in figure 6. Green rays in figure 6 are refraction rays that exist inside objects. Yellow rays in figure 6 are shadow rays.

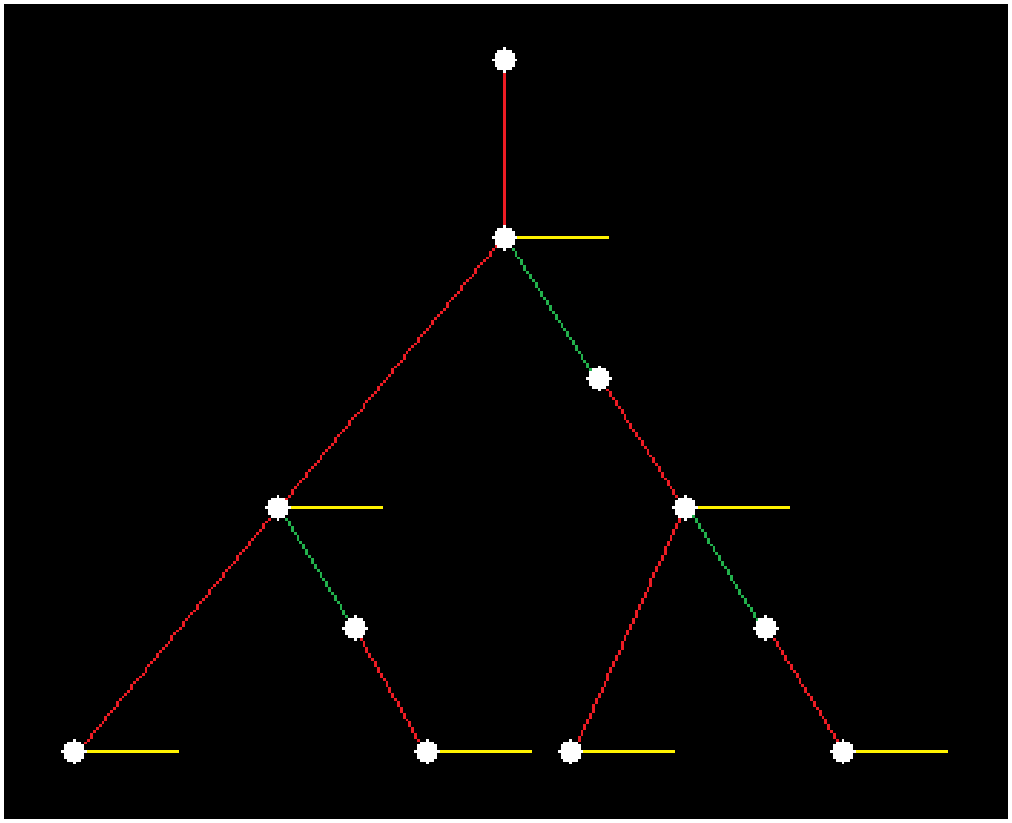


Figure 6: Open air rays are colored red, shadow rays are colored yellow, and rays inside objects are colored green. Study this figure together with figure 2 for a better understanding.

This tree structure is created on the fly in a depth first manner. One start off by looking at the open air root ray coming from the eye. This ray is tested for intersection, if it intersects with an object, and light is visible from this intersection point, illumination is added to the ray's color. Then, if the intersected object can reflect, the ray's reflection value is set to what reflection value the intersected object has, and a new open air child ray is added to the tree structure. This open air child ray is treated in the same manner as the root ray, but the reflection value it got from the object it intersected with, is multiplied by its parents reflection value. This is because all rays need to know how much light hitting the object directly, is remaining after this light has been reflected through all its generations of rays up the tree. When this branch has reached maximum number of generations, or there are no more reflection or refraction rays to add to the bottom of this branch, the last ray that was added to the tree is removed from the three, and this ray's color is added to its parent's color, this happens again and again if there are no more reflection or refraction rays to add to the branch, until the only remaining ray is the root ray. We have then added and removed all rays that derived from the root ray's child reflection ray. A new refraction ray is then looked

at if the object can refract. When this refraction ray exits the objects it was inside, a new open air ray is looked at, this ray is treated equally as all other open air rays. Finally, the execution will terminate and the root ray will contain the correct RGB color for the current pixel in the image plane.

To summarize, in every open air ray, if there is an intersection, a shadow ray is spawned. If the intersected object can reflect, a new reflection ray is spawned and this ray is looked at. When the control finally return to the current ray, a refraction ray is spawned and looked at if the object can spawn a refraction ray. The control flow returns to its parent ray if there are no reflection or refraction rays to add as child rays and all child rays have returned their RGB color, or the limit for maximum number of generations has been reached. To keep track of which rays that are to be added, or removed from the tree, every ray need to have some boolean values that are set to true or false depending on its state.

Review of how light is affected through refraction rays are not described well in this thesis, since this technique will not be implemented, but the framework supporting this phenomenon is described in this section. In this thesis, only reflection and shadow rays are implemented. The structure will then be a linked list, not a tree structure. Since refraction rays, and other rays will be implemented in the future, an algorithm that can spawn and iterate over a tree structure like this, is implemented in hardware.

The easiest way to implement a ray trace algorithm in software, running on a CPU, is probably to implement a recursive algorithm that calls itself every time a new ray is spawned, where the ray returns its color when it terminates. The selected algorithm that is implemented in hardware is a variant of an iterative depth-first post order algorithm. This algorithm is implemented using a stack with a stack pointer. The size of this stack determines maximum number of generations of open air rays (red rays) that are allowed. Only open air rays are stored on the stack. This is because only open air rays can spawn more than one ray, therefore, these rays are the only rays one have reason to visit more than one time, they are therefore stored temporarily on a stack when other rays are visited.

3.6.3 Example: One ray tree iteration

In this example, we will iterate over the tree in figure 7 and observe how the stack and stack pointer behaves.

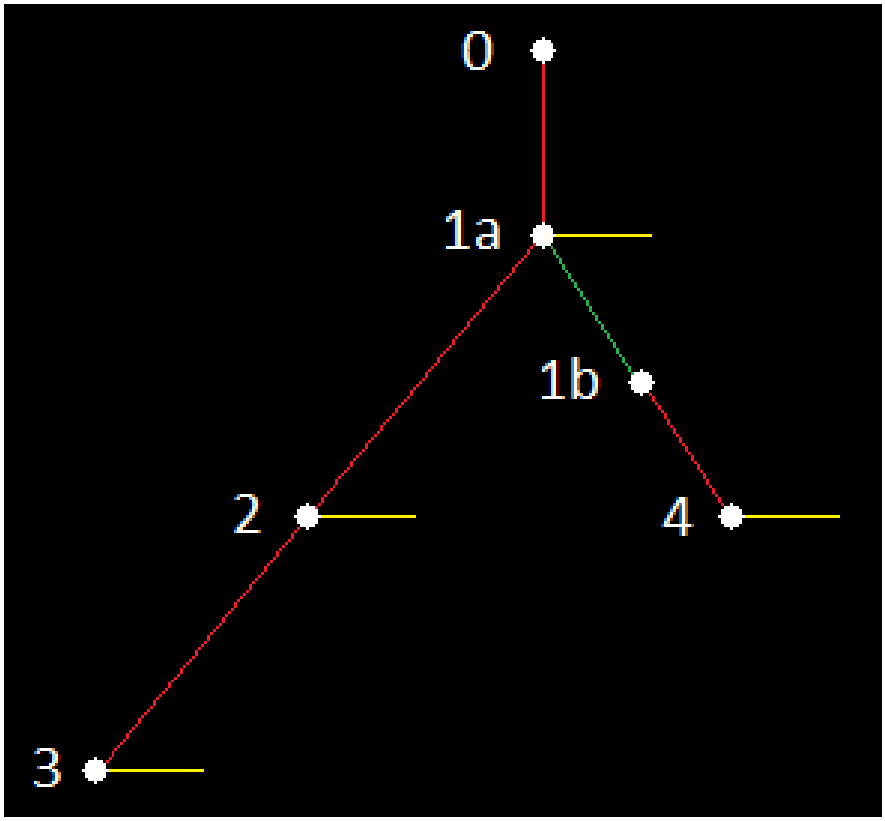


Figure 7: Ray tree: The white dots do not represent objects, they represents intersection points. 1a and 1b are two distinct intersection points on the same object. 2, 3 and 4 are intersection points on unique objects, and 0 represents the eye

Here is how the stack and stack pointer behaves, together with some selected information.

Current Stack pointer	Parent Stack pointer	S0	S1	S2
-	-	-	-	-
S0	-	$0 \rightarrow 1a$	-	-
S1	S0	$0 \rightarrow 1a$	$1a \rightarrow 2$	-
S2	S1	$0 \rightarrow 1a$	$1a \rightarrow 2$	$2 \rightarrow 3$
S1	S0	$0 \rightarrow 1a$	$1a \rightarrow 2$	-
S0	-	$0 \rightarrow 1a$	-	-
S1	S0	$0 \rightarrow 1a$	$1b \rightarrow 4$	-
S0	-	$0 \rightarrow 1a$	-	-
-	-	-	-	-

As we can see, this is not a fully balanced tree. This is because the open air rays that have not reached maximum number of generations may not hit objects that can reflect or refract,

or may not hit objects at all.

3.6.4 Evaluate the algorithm

The selected algorithm is a depth-first postorder iterative tree traversal algorithm. One reason for choosing a depth-first algorithm over a breadth-first algorithm is because in a breadth-first algorithm, all the rays of a generation must be stored in a FIFO queue until their child rays in the next generation can be generated. This can require a large queue if there are a lot of generations of rays, and every generation can spawn many rays like in bidirectional path tracing [2]. When a depth-first algorithm is used, the maximum number of generations of open air rays that are allowed is the same as the size of the stack. By maximum number of generations of open air rays i mean maximum number of open air (red) rays that descend from the eye in the tree in figure 6. A bad feature of depth-first (and perhaps breadth-first) algorithms is that if a scene has many mirror like objects that reflect lights between each other, many generations of rays are required to generate an image with sufficient quality, in this case, one should perhaps consider alternative methods and tricks. The selected scene used in this thesis has many diffuse objects where 4 to 5 generations of open air rays are sufficient, therefore a stack size of 4 to 5 is sufficient when a depth first algorithm is used. How the stack size affect the image in this thesis is described in section 3.8.

3.7 Adjustment of the exposure

As described in section 3.6.1, maximum allowed color intensity is 1.0, and minimum allowed color intensity is 0.0. In some cases the ray tracer can return color values that exceed 1.0. One way to deal with this is to represent every number that exceeds 1 as 1. It is not possible to distinguish e.g., 1.4 and 1.5 when this method is used, since both numbers will be truncated to 1.0. An alternative method that can distinguish color values that exceed 1.0 is to use an exponential function.

$$exposure \in [-\infty, 0] \tag{17}$$

$$color = 1 - e^{exposure \cdot color} \tag{18}$$

The new color value will always be in the range from 0.0 to 1.0. An example of how this function work is shown in figure 8 where the x-aksis represents the old color value that can

range from 0.0 to infinity, and the y-aksis represents the new color value.

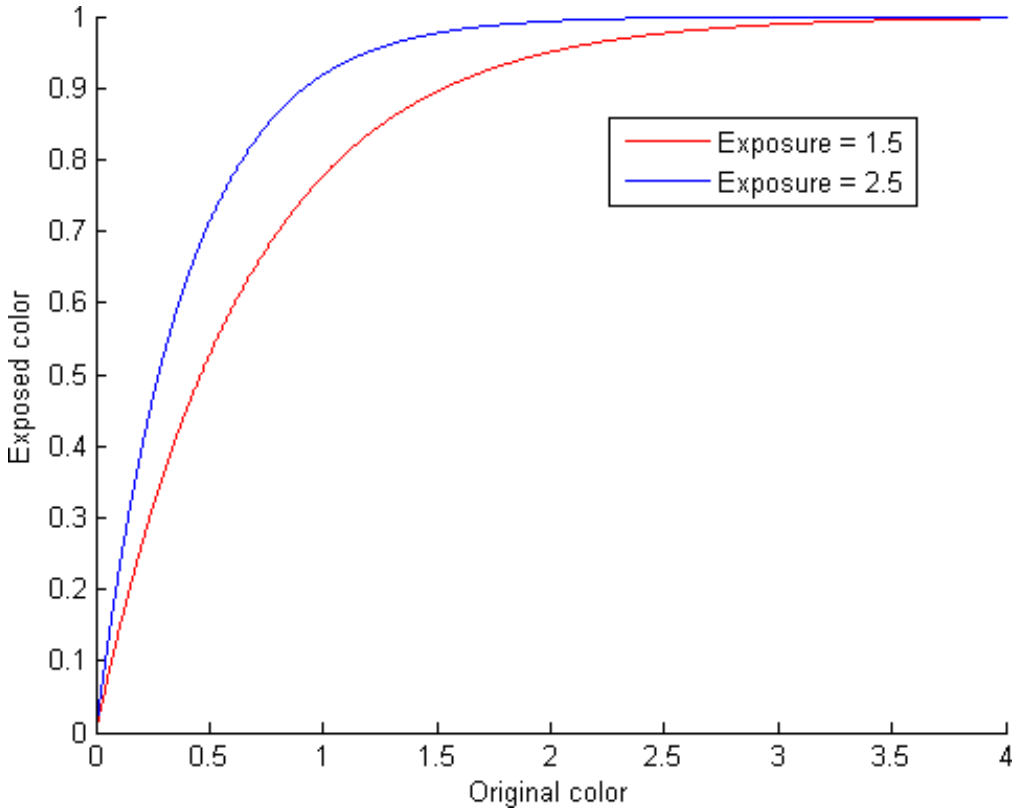


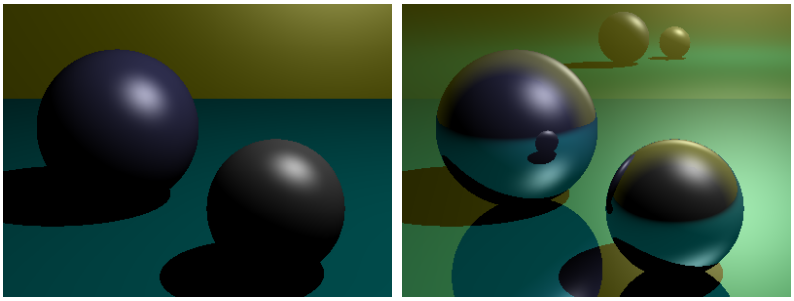
Figure 8: Exposure

In figure 8, a dynamic color range of 0 to 4 is shown. Increasing the exposure value makes it harder to distinguish lower color values, but it makes it easier to distinguish higher color values. Implementing this functionality may require some extra modules on the FPGA that take up space. Therefore, the scene used in this thesis was designed so that a color value above 1.0 will never be returned. If many mirror like objects are used or refraction, global illumination, or other techniques are used, an exponential function may be required.

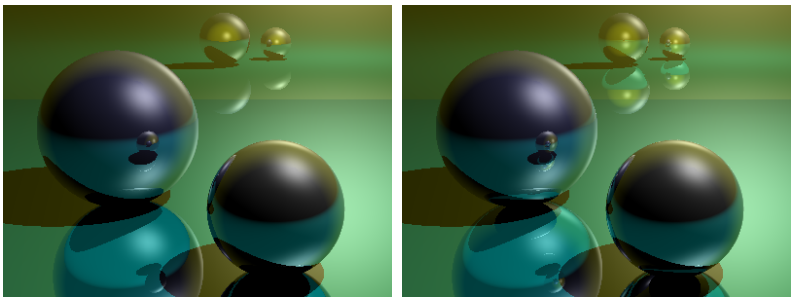
3.8 Selecting a correct stack size

Figure 9 and 10 below show how the stack size affects the resulting image. In figure 9 the planes and spheres has a diffuse value of 0.3 and a reflection value of 0.8. In figure 10 the planes and spheres has a has a diffuse value of 0.5 and a reflection value of 0.4. A higher reflection value implies that more light is reflected from primitive to primitive. To avoid saturation, or use of exponential functions, as described in section 3.7, the planes and

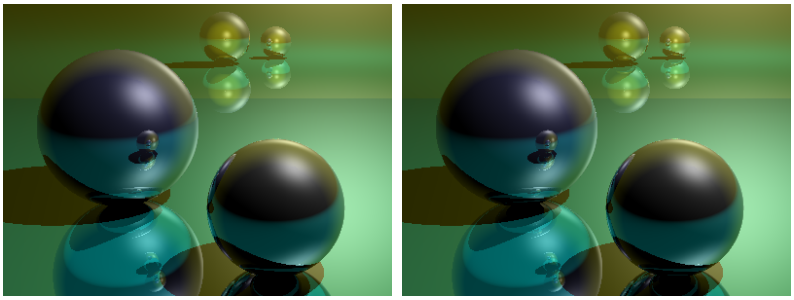
spheres diffuse value must be decreased, so these objects do not scatter to much light coming directly from the light source.



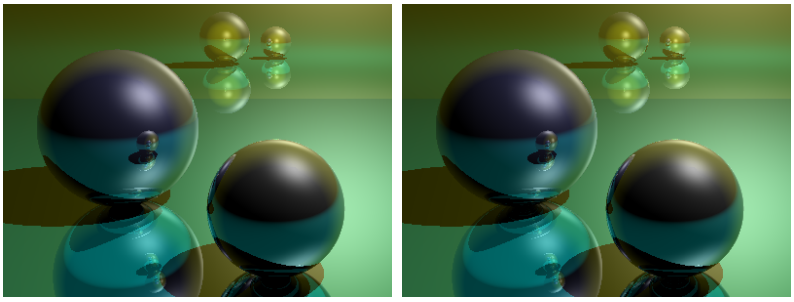
(a) Stack size is 1 (b) Stack size is 2



(c) Stack size is 3 (d) Stack size is 4

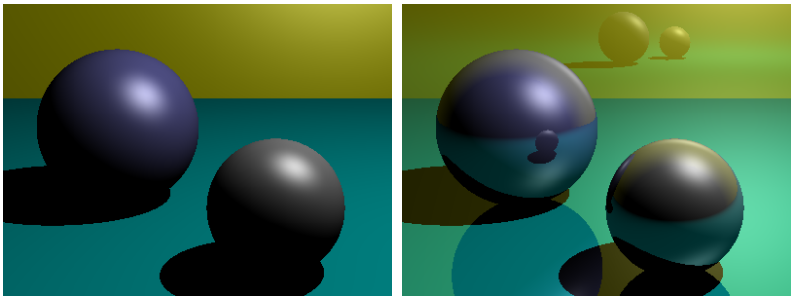


(e) Stack size is 5 (f) Stack size is 6



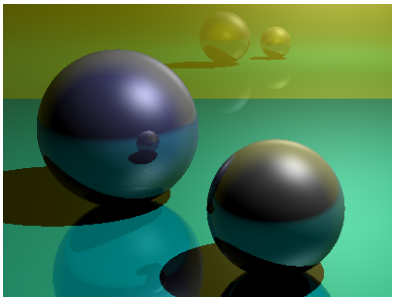
(g) Stack size is 7 (h) Stack size is 49

Figure 9: Reflective surface

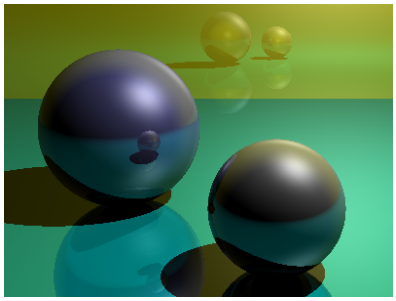


(a) Stack size is 1

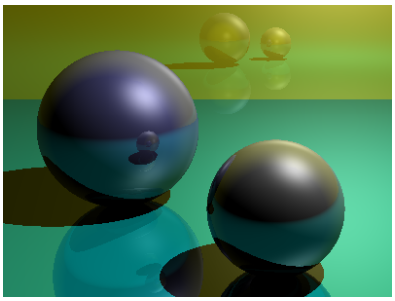
(b) Stack size is 2



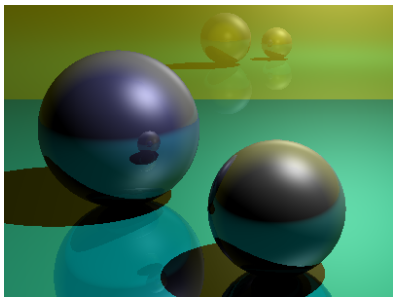
(c) Stack size is 3



(d) Stack size is 4



(e) Stack size is 5



(f) Stack size is 49

Figure 10: Diffuse surface

As we can see in figure 9, a stack size of 5 is sufficient for this reflective scene. In figure 10, which is a diffuse scene, a stack size of 4 is sufficient because the reflective wall in figure 10d and 10e is quite similar, and also very similar to figure 10f which has a stack size of 49. 49 is maximum number of possible reflections in this model, written in C++. In figure 9 we can see that the area facing the camera, between the largest sphere, and the floor, is the area where most reflections occur. Every time a ray hits a reflective surface, the incoming light coming directly from light sources, is multiplied by all its parents reflection values, this new multiplied reflection value quickly decreases to zero, a diffuse scene will therefore normally not require a large stack. The scene in figure 10 is the selected scene with a diffuse value

of 0.5 and a reflection value of 0.4, therefore a stack size of 4 is implemented in Verilog. One trick that can improve the performance of the ray tracer, is not to bother to spawn new reflection and refraction rays if the reflection value has become so small, that the added colors will not make an impression on the final image.

3.9 Estimating the Frame rate

In order to estimate the frame rate, check whether or not it is worth to implement a given functionality in Verilog that may or may not increase the frame rate, check witch Verilog ALU instruction that is most frequently used, and to draw some other conclusions, one must collect some data from the C++ model of the ray tracer. Prototyping in the C++ model is much easier than testing new ideas in the Verilog model, therefore a functionality that keep track of how many times every single floating point operator is used, is implemented in the C++ model. The reason for only collecting information about floating point operators are that the Verilog ALU only work with floating point numbers, or boolean values that are extracted from the floating point numbers. Most boolean and integer operations are done at a higher hierarchical level than the level of the ALU in the Verilog module, these boolean and integer operations may be executed at the same time as floating point operations are executed on the ALU. The operators that are affecting the frame rate performance most is therefore floating point operators, and they are the only operators that are included in the statistics. Section 3.10 below are based on these statistics.

3.10 "Hacks" to increase the Frame rate

Below are some hacks that are implemented in the C++ model, some of these hacks are also implemented in the Verilog model. These hacks are implemented in order to increase the frame rate of the ray tracer, without changing the end result. Note that a dynamic and changing scene will require that some of these hacks must be dynamic, or tuned to fit a specific dynamic scene. The C++ model estimate that the Verilog model will use an average of 246 clock cycles on floating point operations per pixel in the selected scene when none of these hacks are implemented.

3.10.1 Bounding Box

Every time an open air ray is spawned, the new open air ray need to know if it intersects with objects along its path, and it need to know which one of these intersected objects is closest to the rays origin. It is sufficient for a shadow ray to only check if there is an intersection at all along its path to the light source. If no hacks are used, open air rays need to test every object in the scene for intersection, and shadow rays must, in worst case, test every object in the scene for intersection. If there are many objects in the scene, testing for intersection with objects can be a time-consuming task. One method to decrease the number of tests that are done, is to group close objects inside boxes. To verify that a ray does not intersect with a given group of close objects, one only need to verify that the ray does not intersect with the box bounding the given objects. If the ray does intersect with the box, one must check every object inside the box if it intersects with the ray. In most cases, rays will not intersect with a given bounding box, The computation time it takes to check for intersections between rays and objects inside the given bounding box, is then spared in these cases where the rays did not intersect with the given bounding box. This saved computation time is longer than the added computation time resulting from checking for intersection with the rays and the given bounding box. Checking for intersection with a box also takes less time than checking for intersection with a sphere or a plane.

Below are some estimated results showing improvements when bounding boxes are implemented, versus no bounding boxes in the selected scene.

	Average number of clock cycles per pixel that are spent on floating point operations
Without bounding boxes	246
With bounding boxes	205
Reduction	17%

The benefit will increase heavily if a more complex scene is used. Currently only two planes and two spheres are used. If many primitives are used, one could use a hierarchy of boxes.

3.10.2 Fast intersection checking

As described in section 3.3.2. Checking for intersection with planes requires to check if the ray is not parallel to the plane, that is $\vec{n} \cdot \vec{d} \neq \vec{0}$, and to check if the plane is in front of the ray, not behind it, that is $t > 0$. One hack that can be used to check faster for intersection with

planes, is to use the ray direction instead, and compare this direction with the planes normal vector. Since all planes in the selected scene has infinite size, all planes are placed on the edge of the scene, and all planes are aligned along all three axis, it is sufficient to check one vector component, and compare this component against the same component in the planes normal vector. One example of how this is done, is when a plane, that is to be checked for intersection, has a normal vector $\vec{n} =< 0, 0, -1 >$. If the third vector component in the ray direction vector is positive, the plane is facing the ray, the ray will therefore intersect with the plane. This functionality is similar to a bounding box.

Below are some estimated results showing improvements when this functionality is implemented, versus no implementation of this functionality.

	Average number of clock cycles per pixel that are spent on floating point operations
Without fast intersection checking	246
With fast intersection checking	229
Reduction	7%

3.10.3 Efficient checks to see if the light source is visible

It is not necessary to check if planes are blocking the light source since planes are, in this thesis, placed on the edge of the scene. It is not necessary to compute the distance from an intersection point to the light source when spheres are tested for intersection with shadow rays, since spheres will never be placed on the opposite side of the light source, and planes are never tested for intersection with shadow rays. It is therefore sufficient to check if shadow rays intersect with spheres when one need to determine if an intersection point is visible from the light source. Therefore a function that returns the length of a vector is not needed, nether in the rest of the code. Equation 19 shows how this could be done. Implementing a bounding box is also relevant in this context. The light source in this scene is infinitesimal and is placed outside the field of view.

$$t = \sqrt{v_x \cdot v_x + v_y \cdot v_y + v_z \cdot v_z} \tag{19}$$

	Average number of clock cycles per pixel that are spent on floating point operations
Without efficient visibility checks	246
With efficient visibility checks	220
Reduction	11%

3.10.4 Using all three hacks

Every hack listed above are activated.

	Average number of clock cycles per pixel that are spent on floating point operations
Without any hacks	246
With all hacks	162
Reduction	34%

3.10.5 Hack combination used in the Verilog module

Every hack listed above are activated, except for bounding boxes. It is easy to implement bounding boxes in the future.

	Average number of clock cycles per pixel that are spent on floating point operations
Without any hacks	246
With two hacks	203
Reduction	17%

3.11 Performance results

The C++ ray tracer is tested on a virtual machine running Ubuntu. The virtual machine has been granted 1233 megabytes of RAM and one of two CPU cores from Intel. The CPU is an E8400 Core 2 Duo running at 3.00GHz. 2.56 frames per second was achieved. 2.56 FPS is equivalent to having a frame period of 0.39.

3.12 Experiences regarding prototyping in C ++

Below are some experiences that i have made regarding modeling of the ray tracer in C++. The purpose of the C++ model is to get theoretical knowledge and practical experience regarding ray tracers, and to use the C++ model in the design and verification process of the Verilog model. These tips and techniques may not be universal, but i found them very useful when i was converting the C++ ray tracer to a Verilog ray tracer.

Recursion VS Iteration

Since the C++ model is used in the design and verification process of the Verilog model, a close match between the C++ model and the Verilog model is desired. Recursion in C++ is when a function calls itself. In C++ when a function calls it self (or another function), the state of the functions caller must be preserved. This state is stored on a stack. Handling of the stack and the stack pointer when recursion is used, is in some degree hidden for the programmer. An iterative tree traversal algorithm with a stack and a stack pointer is implemented in C++ model instead of a recursive algorithm. In an iterative approach, the control flow and overhead is more explicit. It is then easier to convert the C++ model into a Verilog model, it is easier to design and debug since handling of the stack and the stack pointer is now explicit.

Map reusable code into functions

Similar C++ codes that are used more than one time may be mapped into functions. If functions are used is it easier to spot which code blocks that can be modeled as sub state machines in Verilog. A function modeled in C++ may take parameters and return values.

Use fewest possible state variables

It is vital that rays contain minimum number of state variables. This is because the stack can contain many rays. If one ray increase in size, the stack need to increase this size times the number of rays that a stack can contain.

Limit the duration a value must be accessible

This will decrease the number of temporary registers required in the Verilog model.

One Verilog ALU instruction per line of C++ code

If possible, write C++ code so that one line of C++ code can be converted to one Verilog ALU instruction. This way it is easier to convert the C++ code to a state machine with instructions, and it is easier to tweak the code to get fewer Verilog ALU instructions. One example of this is when the C++ code that rotates rays coming from the eye, is converted to Verilog code. It is easy to experiment with the C++ code on how the Verilog ALU instructions should be arranged to decrease the number of clock cycles that are used. This creates a tight match between the C++ and Verilog code, it is therefore easier to design and verify in Verilog. Some knowledge of the ALU must be known in order to optimize. Optimization should be done before the Verilog ALU core is built, and along until the design of the Verilog modules are finished.

Keep track of number of times operators are used

If it is known how many Verilog ALU clock cycles every operator need, is it easy to estimate how much improvement one tweak or added functionality will imply. Therefore is it wise to keep track of the number of clock cycles that are used in the Verilog module to find out if a tweak or functionality is worth to implement.

4 Implementation in Verilog

When i wrote the C++ model of the ray tracer, i figured out that it would be best to write a processor that has pixel coordinates as input, instead of rays as input. A processor with rays as input must be able to support many types of rays, some unknown rays that are to be implemented in the future. I did not know how this would affect the processor, so i went for a processor that would work on pixels. Then the question came if i wanted to pipeline the pixel processor, or if i wanted to have many parallel pixel processors. After doing some hand calculations, i figured out that a fully pipelined pixel processor would probably not fit on the FPGA since floating point operations are done instead of fixed point operations. Floating point operators are known to take up a lot of space on FPGAs. The negative side is that the utilization will be low, which then will lead to lower FPS.

Since componentwise vector multiplication and vector addition/subtraction is frequently used, a vector ALU is built. Vector operations take then just as long time as scalar operations, but the operators occupy three times as much area on the FPGA. A figure of the top module that can contain one or more pixel processor is shown in figure 11. This top module is actually not implemented as a synthesisable module, it is rather implemented to some extent as a testbench in Verilog. The top module in figure 11 is only here to put things in perspective, not to give an exact representation of how it should be implemented in Verilog.

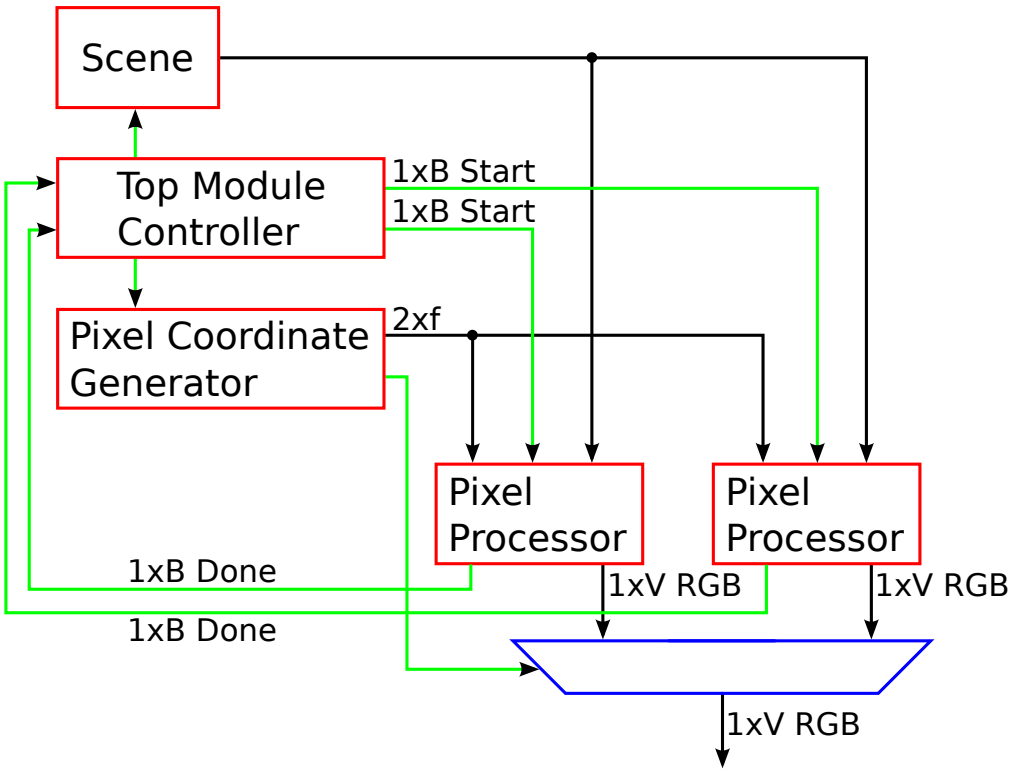


Figure 11: Top module. Green signals are control signals. The clock and reset signal are not included in this figure

4.1 Overview

The top module contains one or more pixel processors. Each pixel processor contain a pixel processor controller and a pixel processor core. Each pixel processor core contain four modules, namely the ALU, ALU Controller, Memory and Memory Controller. 32 bit floating point numbers seemed to be sufficient in the C++ model, 32 bit floating point numbers are therefore used in the Verilog model.

4.2 ALU

The ray tracer model requires heavy use of floating point operations. These floating point operations are computed by floating point operators, these floating point operators can consume a lot of area on the FPGA. It is therefore important to reuse as much hardware as possible, to get ideally close to 100% utilization. Therefore an arithmetic logic unit (ALU) that only operates on floating point numbers, is created. Integers and boolean values that do

not relate to floating point numbers, are handled at a higher hierarchical level, at the "pixel processor controller" level, in order to save clock cycles. It is also of interest to create a clear boundary between where floating point operations are done, and where boolean and fixed point operations are done. Far from 100% utilization is achieved, but reuse of floating point operators throughout one iteration of the pixel processor, is done. Floating point addition and multiplication operations are often done on vectors which have three components each, one vector addition and one componentwise vector multiplication operator is therefore made. Floating point reciprocal (multiplicative inverse) and square root operations are often done on scalars, so these operators are not made as vector operators, they are made as scalar operators.

4.2.1 Major ALU components

Vector addition This is a combinatorial circuit.

$$\begin{aligned} \vec{v} &= \vec{v_1} + \vec{v_2} \\ < v_x, v_y, v_z > &=< v_{1x} + v_{2x}, v_{1y} + v_{2y}, v_{1z} + v_{2z} > \end{aligned} \tag{20}$$

Componentwise vector multiplication This is a combinatorial circuit. The selected notation for componentwise vector multiplication is

$$\begin{aligned} \vec{v} &= \vec{v_1} * \vec{v_2} \\ < v_x, v_y, v_z > &=< v_{1x} \cdot v_{2x}, v_{1y} \cdot v_{2y}, v_{1z} \cdot v_{2z} > \end{aligned} \tag{21}$$

Notice that the notation for scalar multiplication is $v_{1x} \cdot v_{2x}$ not $v_{1x} * v_{2x}$.

Float Reciprocal ($\frac{1}{x}$) Consumes four clock cycles. This function block is pipelined, but the pipeline functionality is not needed.

$$v_x = \frac{1}{v_{1x}} \tag{22}$$

Float square root Consumes 11 clock cycles. This function block is pipelined, but the pipeline functionality is not needed. This function block does also contain a lot of functionality like exponential and sinus functions. They are not removed manually, but the synthesis tool remove some of this functionality. This intellectual property is only used as a temporary

solution.

$$v_x = \sqrt{v_{1x}} \tag{23}$$

Changing the signed bit This is a combinatorial circuit. All three signed bits in vector 1 can be altered. This functionality can only be used together with vector addition and vector multiplication. Vector addition and vector multiplication have their own signed bit altering functionality that can be controlled individually. The custom function described in section 4.7.1 requires that multiplication and addition are performed at the same clock cycle, with opposite sign on one of their inputs.

$$\begin{aligned} \vec{v} &= -\vec{v_1} \\ < v_x, v_y, v_z > &=< -v_{1x}, -v_{1y}, -v_{1z} > \end{aligned} \tag{24}$$

Bypass vector This is a combinatorial circuit.

$$\vec{v} = \vec{v_1} \tag{25}$$

Below is a figure that illustrates how the ALU is built.

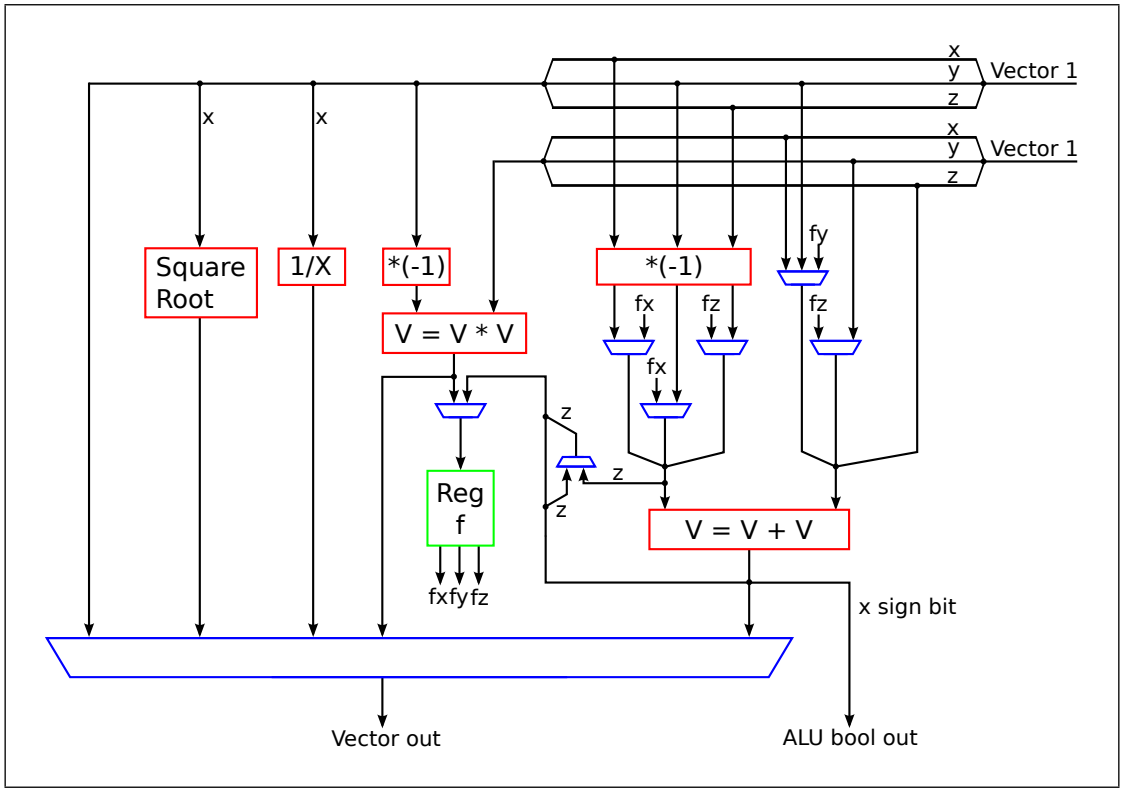


Figure 12: ALU. Only data path is shown.

4.3 ALU controller

An ALU controller is built. This controller takes an instruction as an input value, and combinatorially control the ALU, depending on the instruction word it receives. Some ALU operators are multi cycle, these multi cycle operators read and write hand shake signals from and to the pixel processor controller, these operators also receive control signals from the combinatorial ALU controller.

4.3.1 ALU controller instructions

The ALU controller's only input is the instruction word input. This instruction word sets every control signal in the ALU, except for the "multistage input valid" signal that is used for multi cycle ALU operators. This signal is set explicit by the pixel processor controller's state machine. This state machine must therefore set the desired instruction, and the "multi stage input valid" signal to true, and then wait in the next state for the "multi stage output

valid" signal to become true, this signal is true when the output data are ready. The selected instruction must be the same when the output valid signal is true, as when the input valid signal was true, in order to put the ready data on the ALU output. The reason for triggering the input valid signal from the pixel processor controller's state machine, outside the ALU controller, is because it is desired that hand shake signals should be read and written at the same hierarchical level as instruction words are written. The instruction words that are supported by hardware are one single instruction for every single cycle operation, and one single instruction for every clock cycle in every multi cycle operation, except for reciprocal and square root operations, because these multi cycle operations use hand shake signals. These two multi cycle operations have therefore only one instruction word each, their instruction word must be set at the beginning and at the end of the operation.

4.4 Memory

The memory module contains some memory locations used for temporary storage of floating point numbers, and one stack that can contain rays. The scene is forwarded from the top module since the same scene is used by every pixel processor. Writing is done synchronously at the rising edge of the clock, reading is done asynchronously as a combinatorial circuit on the output of the memory module. Keeping reading and writing operations synchronously instead, is probably a good idea, since RAM resources on the FPGA can be used, instead of LUTs, as described in section 5.1. The memory module can read from two locations and write to one location at the same time. This also applies when reading and writing are done at the same time to the same memory location. The memory module receives decoded write and read addresses from the memory address decoder.

A vector consists of three floats. If a vector is addressed, every float component inside this vector is addressed together. Some vectors allow for their components to be addressed individually. This is an important feature, especially in the case where an Euclidean vector is rotated as in section 3.5 where only one of three vector components is updated.

4.5 Memory controller

Input signals to the memory controller are stack pointer, one write address, and two read addresses. These signals are coming from the pixel processor controller. One decoded write address, and two decoded read addresses are generated combinatorially, by the memory

controller, depending on the state of the three input signals. These decoded addresses are set as output signals from the memory controller module, and input signals to the memory module.

Instead of addressing rays in the stack directly, one can address the current ray, and the parent ray. These addresses are decoded combinatorial to what address that holds the current and parent ray, all other addresses remain the same.

4.6 Pixel processor core

The pixel processor consists of all these four modules mentioned above. How all these four modules are glued together, is shown in figure 13. Only data path signals are shown. The next module in the immediate level above this pixel processor core, is a pixel processor, containing a pixel processor core, and a pixel processor controller. All these signals that are connected two and from the pixel processor core in figure 13 are signals that communicate and exchange data to and from the pixel processor controller, except for the Scene input signal and the Stack 1 RGB output signal. These signals are forwarded from, and sent to an even higher hierarchical level, to the topmodule.

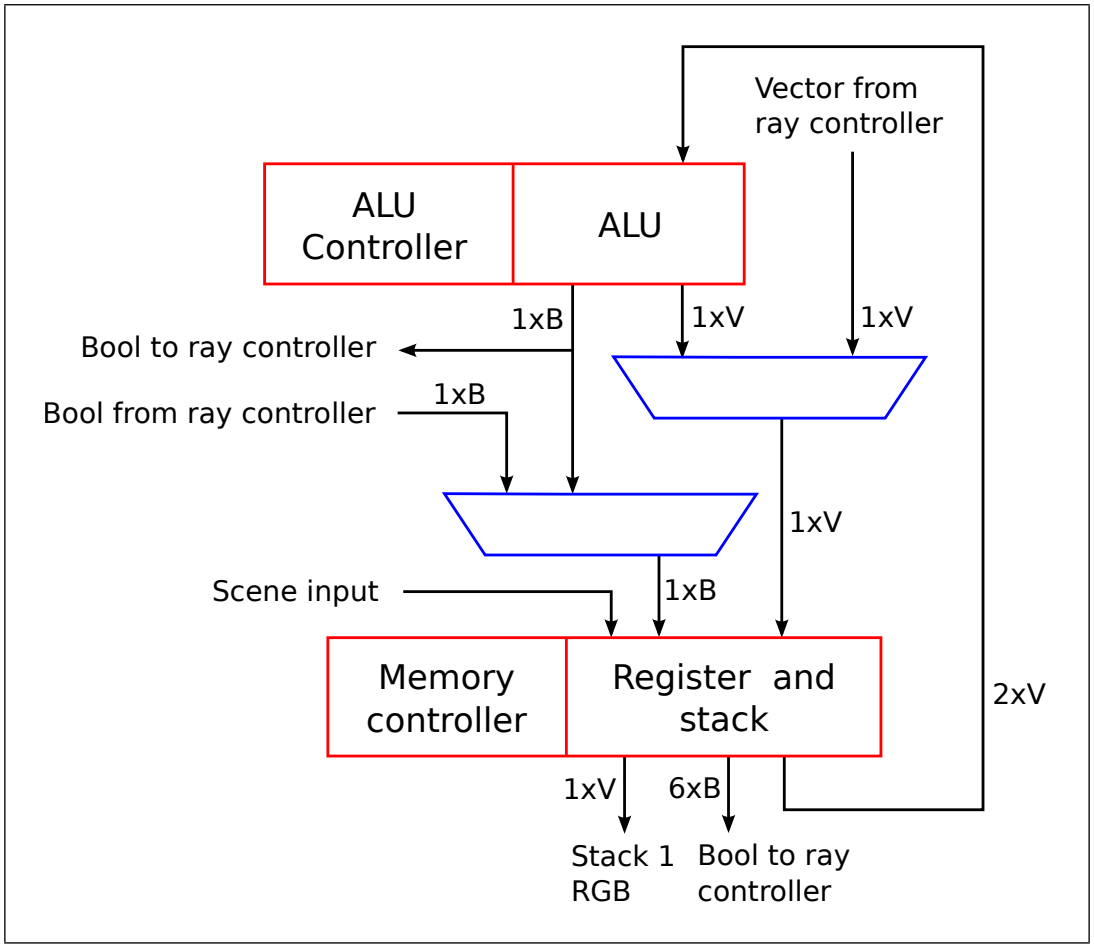


Figure 13: Pixel Processor. Only data path is shown.

4.7 Pixel processor controller

The goal for this pixel processor controller is to control one pixel processor core. The controller has only one single large state machine where some of these states have designated roles as reusable sub state machines, or subroutine. The pixel processor controller is the Verilog module that has most similarities to the C++ models functions (or methods), since one state in this controller can map to one line of C++ code. The input to this model are two pixel coordinates and a start signal. Some other signals also exist that communicate, and send data to and from the pixel processor core, and one done signal that is sent to the top-module when the pixel processor core finishes its execution on one pixel. The stack pointer is stored in this model, the stack pointer can be incremented and decremented synchronously. There are also seven pointer registers, these registers are used in reusable sub state machines.

One register holds a return value, this register is equivalent to the boolean value a function in C++ can return. One other state variable is also stored in this module, this state variable can contain an ID of a primitive, this state variable is used e.g., when one need to know which primitive intersect, and is closest to the ray origin. A C++ function that can pass by reference, and return a boolean value, is equivalent to a sub state machine in Verilog that work on registers pointed to by pointer registers in beforehand, and set a boolean value in a return register when the sub state machine finish its execution.

4.7.1 Pixel processor controller's sub state machines

These following sub state machines are Verilog versions of frequently used functions in the C++ model that do not translate to one single cycle ALU instruction, except for float square root and float reciprocal.

Dot product The dot product consists of one componentwise vector multiplication and two float additions. This function consumes three clock cycles.

$$\begin{aligned} v_x &= \vec{v_1} \cdot \vec{v_2} \\ v_x &= \vec{v_{1x}} \cdot \vec{v_{2x}} + \vec{v_{1y}} \cdot \vec{v_{2y}} + \vec{v_{1z}} \cdot \vec{v_{2z}} \end{aligned} \tag{26}$$

Normalize vector This function consists of one dot product, one float square root, one float reciprocal, and one componentwise vector multiplication. This function consumes 19 clock cycles.

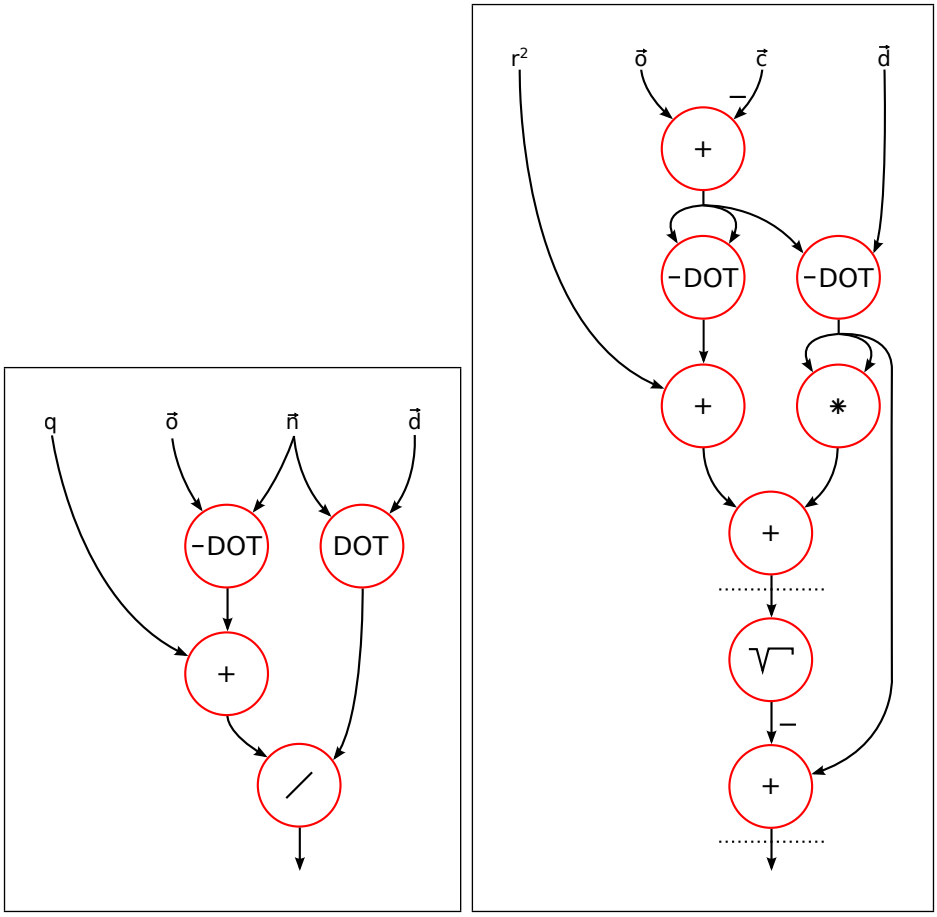
$$\vec{v_1} = \frac{\vec{v_1}}{\sqrt{\vec{v_1} \cdot \vec{v_1}}} \tag{27}$$

Sphere - GetNormal This function consists of one vector subtraction (vector addition where the signed bit is inverted in every component in one of the vectors), and one vector multiplication. This function consumes two clock cycles.

$$(pointOf\vec{I}ntersection - sphere\vec{C}enter) * radiusDevided \tag{28}$$

Every spheres $radiusDevided$ value, that is, $\frac{1}{radius}$, are stored in memory. This is because multiplication takes up less clock cycles than division which consists of one reciprocal operation and one multiplication operation.

Plane and sphere - Intersection A large portion of the ray tracers execution time is spent on checking whether or not a ray is intersecting with any planes or spheres. Since an ALU architecture had already been seriously considered, effort was devoted to check if customizing the ALU hardware could improve the performance of these intersection functions. I first set up a data flow graph for the sphere intersection function in equation 4 to see if i could increase the utilization of the ALU operators, which was possible to do. I then made a data flow graph of the plane intersection function in equation 8. I found that if i implement the plane intersection hack as described in section 3.10.2, i do not need to check for conditions whether a ray does or does not intersect with a plane. I could then do the same utilization in the plane intersection function, as in the sphere intersection function, and i found the common sub graph that covered both these improvements. The two data flow graphs are shown in figure 14.



(a) Plane Intersect (b) Sphere Intersect

Figure 14: Data flow graph of intersect functions

The result from the dot operation $(\vec{o} - \vec{c}) \cdot \vec{d}$ in figure 14b is multiplied by -1. It is then necessary to only invert one of the input values in the last addition operation. The Verilog ALU can only invert one input value at a time.

The common sub graph mentioned above, is shown in figure 15.

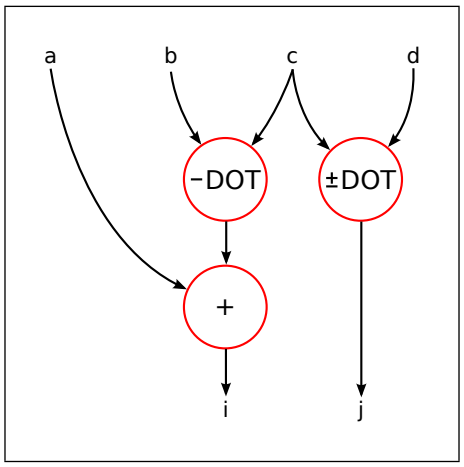


Figure 15: Common sub graph

This common sub graph is implemented as a sub state machine in the pixel processor controller. The goal for this sub state machine is to generate the following result with fewest possible clock cycles, and reuse chosen ALU blocks like vector addition and vector multiplication.

$$\text{Result} =< -b_x \cdot c_x - b_y \cdot c_y - b_z \cdot c_z + a, \pm c_x \cdot d_x \pm c_y \cdot d_y \pm c_z \cdot d_z, DK > \tag{29}$$

DK means Don't Care. As described in section 4.2.1, vector addition and vector multiplication takes up one clock cycle each. Dot product consists of one vector multiplication and two float additions. There is a total of two dot product operations and one addition operation in figure 29 that can be done serially on the ALU. The total number of clock cycles would therefore be 7 clock cycles if this had been done serially. Equation 29 can be done at minimum 4 clock cycles with the chosen ALU blocks. Below is how this function is implemented

in hardware.

Clock cycle	Output	ALU instruction	Input vector 1	Input vector 2
1	$\vec{f} =$	$-\vec{v_1} * \vec{v_2}$	$\vec{v_1} = \vec{b}$	$\vec{v_2} = \vec{c}$
2	$\vec{g} =$	$\pm\vec{v_1} * \vec{v_2}$	$\vec{v_1} = \vec{d}$	$\vec{v_2} = \vec{c}$
	$\vec{f} =$	$< f_x + f_y, DK, f_z >$		
3	$\vec{f} =$	$< v_{1x} + v_{1y}, f_x + f_z, v_{1z} >$	$\vec{v_1} = \vec{g}$	$\vec{v_2} = \vec{DK}$
4	$\vec{f} = \vec{g} =$	$< v_{1x} + f_y, f_x + f_z, DK >$	$\vec{v_1} =< a, DK, DK >$	$\vec{v_2} = \vec{DK}$

(30)

$\vec{f}$ is an internal register inside the ALU and $\vec{g}$ is an external register outside the ALU. The reason for using an internal register is because, in clock cycle 2 in equation 30, one need to read from two vectors and two floats ($\vec{v_1}$, $\vec{v_2}$, f_x and f_y), and write to one vector and one float ($\vec{g}$ and f_x). The ALU can only read from two vectors, and write to one vector in one clock cycle, this is because a clean interface to the ALU is desired. The internal register was set to have the size of a vector, not two floats. If the internal register had the size of two floats, the function would have required, at least, an extra float in the external register. How this can be done is shown in equation 31.

Clock cycle	Output	ALU instruction	Input vector 1	Input vector 2
1	$\vec{f'} =$	$< -v_{1x} \cdot v_{2x}, -v_{1y} \cdot v_{2y} >$	$\vec{v_1} = \vec{b}$	$\vec{v_2} = \vec{c}$
	$\vec{h} =$	$< DK, DK, -v_{1z} \cdot v_{2z} >$		
2	$\vec{g} =$	$\pm\vec{v_1} * \vec{v_2}$	$\vec{v_1} = \vec{d}$	$\vec{v_2} = \vec{c}$
	$\vec{f'} =$	$< f'_x + f'_y, DK >$		
3	$\vec{f'} =$	$< v_{1x} + v_{1y}, f'_x + v_{2z} >$	$\vec{v_1} = \vec{g}$	$\vec{v_2} = \vec{h}$
4	$\vec{f'} =$	$< v_{1x} + f'_y, f'_x + v_{2z} >$	$\vec{v_1} =< a, DK, DK >$	$\vec{v_2} = \vec{g}$
	$\vec{g} =$	$< v_{1x} + f'_y, f'_x + v_{2z}, DK >$		

(31)

$\vec{h}$ is here an external memory location for a vector (alternatively a float). f' means that the internal register has only space for to floating point numbers. The difference between equation 30 and 31 is that in clock cycle one in equation 31, the result is stored in the external and internal register instead of only the internal register. The $\vec{v_2} = \vec{h}$ in clock cycle 3 in equation 31 is used to compensate for the loss of internal register size. There are also some other differences that are not relevant to this discussion.

4.8 Pixel processor

The only purpose of this module is to glue together the pixel processor controller, and the pixel processor core. The input signals to this module are two pixel coordinates and a start signal that are forwarded to the pixel processor controller, and the scene that is forwarded to the pixel processor core, and then to the memory module. The output signals are the done signal that is forwarded from the pixel processor controller and the stack 1 RBG signals that are forwarded from the memory module, via the pixel processor core. The stack 1 RGB signals provide the final RGB color that the selected pixel should have on the image plane.

4.9 Topmodule

This top module is yet to be implemented in Verilog. It should contain the scene, a generator that generates pixel coordinates, several pixel processors, and a controller that can distribute tasks to every pixel processor, and forward their RGB color to the display, and probably alter the scene. At the moment, a test bench does some of this in order to test and verify the pixel processor.

4.10 Verification process

Effort was made to automate the verification process and to give feedback if the system had a failure. The feedback consist of information on e.g., roughly where the fault is and how large the error is. This can be done because the C++ and Verilog model are printing some floating point variable and register values to their own file. The C++ and Verilog model are supposed to print equivalent floating point information to a new line in their own file at the same state. A tool i have written in C++ compare these two files and report if two equivalent lines have floating point numbers that differ to much in value. A more advanced technique is used to find the difference between two floating point numbers. This technique gives the difference in number of units in the last place (ULP). If there is a difference that exceeds a maximum value, relevant information is printed on the computer screen, this information is e.g., roughly where in the C++ and Verilog code the fault is. It is not necessary to update the testbench if the input to the pixel processor has changed. Many other tools that automate the verification process has also been made, but they are not mentioned in this report.

4.11 Simulation results

The number of clock cycles it takes for one single RTL pixel processor to generate an 800x600 image plane, is 148.655.803. This is an average of 310 clock cycles per pixel. Every RGB color value that returned from the Verilog simulator, was stored to a file. The C++ model is using some functionality from Qt [3] that can display this data. In figure 16, a visual representation of the data returned by the Verilog simulator is shown. The C++ model estimated that an average of 203 clock cycles would be spent on floating point operations. One can then conclude that about 35% of the time is spent doing other things than floating point operations, if the selected scene is used, and the C++ models estimate is correct.

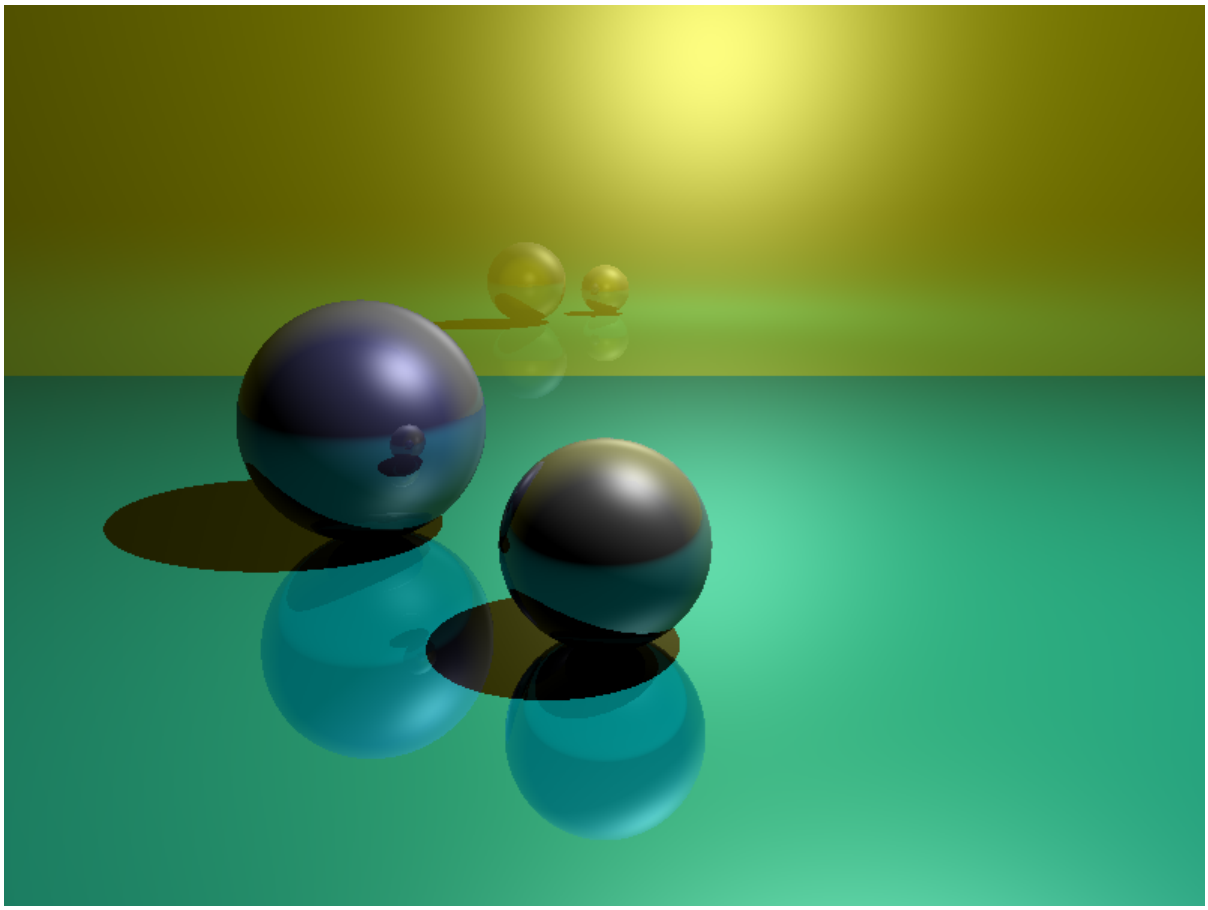
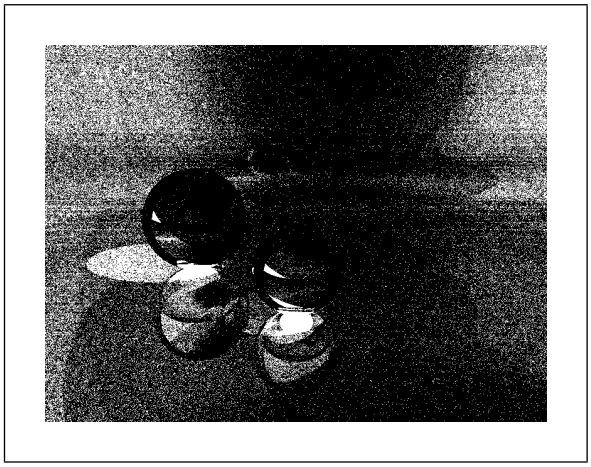


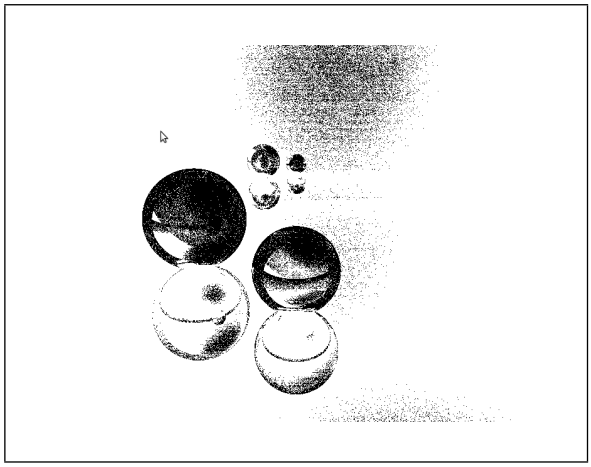
Figure 16: Visualization of data returned by the Verilog simulator. Image size is 800x600 pixels.

It is not easy to compare visually this image with the image generated by the C++ Ray tracer model. Images that display the difference between the C++ model, and the Verilog

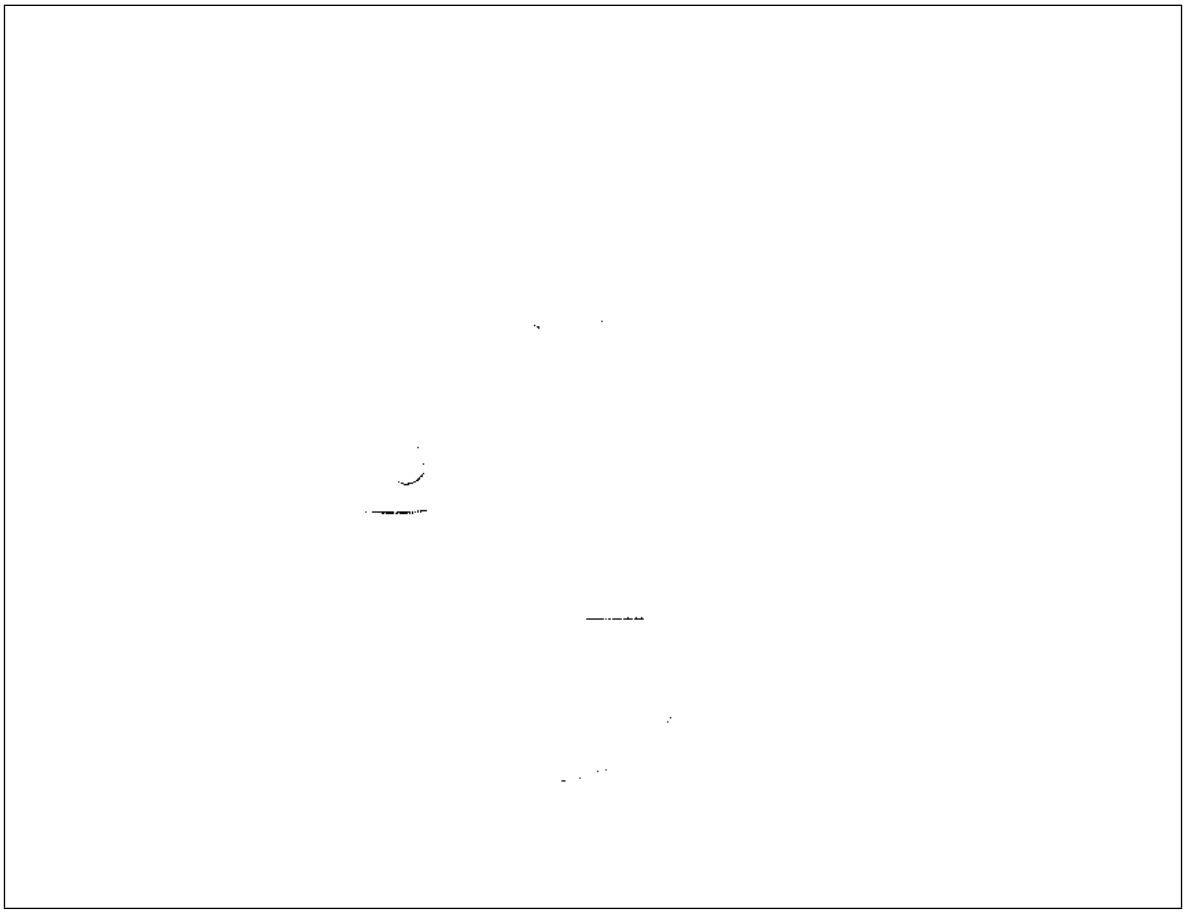
model, in a better way, are shown in figure 17. The absolute value of the difference between the C++, and Verilog modules 800x600 RGB color values, are checked to see if these values exceed a maximum value. If the difference is greater than this maximum value, a black dot is printed in this location. Read, green, and blue color values, range from 0 to 255, where 255 is maximum presence of a color, and 0 is minimum presence of a color. In section 3.6.1, this range is from 0 to 1, so every color value must be multiplied by 255 to scale to this new range.



(a) Absolute value of difference > 0.00001



(b) Absolute value of difference > 0.0001



(c) Absolute value of difference > 0.5

Figure 17: Comparison of RGB values returned by the C++ and Verilog models

The table below shows how many percent of all color differences exceeded the given limit, and what the average difference of all these colors that exceeded this limit is.

Maximum acceptable difference	Percentage of colors that exceeds this limit	Average difference for every color that exceeds this limit
0.00001	62%	0.00148
0.0001	9%	0.01003
0.5	0.03%	2.56894

The three largest differences are 18, 11 and 9. The ray tracer is written for a display that can distinguish 256 different versions of every red, green and blue color. Every color value that has a decimal number of 5 or more, will then be rounded up to the nearest integer, every color value that has a decimal number lower than 5 will be rounded down to the nearest integer. If the C++ model is viewed as a golden version, a difference less than 0.5 would be acceptable. In figure 17 we see that color differences that exceed this limit is only in places where the root ray, or its child rays barely, or barely not intersect with a primitive. This will not spoil the visual effect since, in real life, some noise exist. Noise in the Verilog and C++ model is therefore not seen as a major drawback.

The reason for why there is a difference between the C++ model and the Verilog RTL model, is because different floating point operators have varying degrees of accuracy. The accuracy of floating point operators are given in number of "units in the last place" (ULP). The accuracy can differ from operator to operator. To get more information about floating point arithmetic one can read the IEEE Standard concerning Floating-Point Arithmetic [1].

5 RTL Synthesis

My supervisor Øystein Gjermundnes had a prototype board from Xilinx with a Virtex6LX760 FPGA. This FPGA is on of Xilinx largest FPGAs in their sixth series of FPGAs [4]. Xilinx ISE was the selected tool for doing synthesis and analysis of the Verilog design. Default process options in Xilinx ISE were selected, the only constraint that was set, was that a clock frequency of 50MHz is desired. Since every pixel processor is running independently and parallel to all other pixels processors, the total amount of clock cycles it takes for one pixel processor to compute one complete frame is divided by how many pixel processors there are in the design.

5.1 Synthesis messages

The synthesis tool reported that some RAM blocks are implemented on LUTs because these ram blocks have asynchronous read. If these ram blocks have synchronous read, one can take advantage of available block RAM resources for optimized device usage and improved timings.

5.2 Results and Discussion

There were not used any latches of any kind in this design. The results returned by Xilinx ISE, and some other hand calculated results, are as following.

1 Pixel Processor	(Used/Available/Utilization)
Number of Slice Registers (Flip Flops)	3501/948480/0.4%
Number of Slice LUTs	13837/474240/2.9%
Maximum frequency	34.740MHz
FPS	0.234
Frame period (1/FPS)	4.279s

2 Pixel Processors	(Used/Available/Utilization)
Number of Slice Registers (Flip Flops)	6992/948480/0.7%
Number of Slice LUTs	27419/474240/5.8%
Maximum frequency	28.862MHz
FPS	0.388
Frame period (1/FPS)	2.575s

4 Pixel Processors	(Used/Available/Utilization)
Number of Slice Registers (Flip Flops)	≈14227/948480/1.5%
Number of Slice LUTs	≈5216/474240/11.0%
Maximum frequency	n/a
FPS	n/a
Frame period (1/FPS)	n/a

When a total number of four pixel processors was compiled by Xilinx ISE, the tool aborted the place and rout process because it could not route the remaining 218 signals within a reasonable amount of time. Most of these signals are signals that are related to the memory module. Xilinx ISE also reported that the router had detected a very congested design. Since only 1.5% of every slice register were used, and only 11.0% of every slice LUT were used, it would have been sufficient for a router to route this design if the design was not so congested, since it is a lot of space left on the chip that can be used for routing.

In the case where only 2 pixel processor were used, the tool reported that in one of the longest paths, only 7.9% of the path was consumed by logic, the remaining 92.1% of the path was consumed by routings. This is another indication of that there should be less routing to decrease the longest path that in turn will increase the clock frequency which gives a higher FPS, and make sure that more pixel processors can fit on the FPGA.

Almost at the beginning of one of the longest paths, generated by the state in figure 18, a read address is set for one memory vector output. The resulting output determines what read address one get for the second memory vector output. These two vectors are transported into the vector adder block inside the ALU. The result from the vector adder is then stored in a memory location. This combinatorial path goes consecutive through two large memory output muxes. This is not very convenient way to do things, in respect of maximum clock frequency. It is not wise to implement to much consecutive combinatorial logic in the pixel processor controller's state machine. A solution to this problem is shown in figure 19.

```
RAYSTATE_ITERATIVE_30: begin
  read_address_1_r    = `ADDRESS_REG_FLOAT1;
  if (memory_bool1_out[0] === `FALSE) begin
    read_address_2_r  = `ADDRESS_REG_FLOAT1;
    alu_instruction_r = `INSTRUCTION_ADD;
    write_address_r   = `ADDRESS_REG_FLOAT1;
    write_enable_r    = `TRUE;
    ray_state_nxt     = RAYSTATE_ITERATIVE_31;
  end
  else begin
    ray_state_nxt     = RAYSTATE_ITERATIVE_PUSH_1;
  end
end
```

Figure 18: Consecutive combinatorial logic in the pixel processor controller's state machine, resulting in a longer path. Every signal has a default value.

```
RAYSTATE_ITERATIVE_30: begin
  read_address_1_r  = `ADDRESS_REG_FLOAT1;
  read_address_2_r  = `ADDRESS_REG_FLOAT1;
  alu_instruction_r = `INSTRUCTION_ADD;
  write_address_r   = `ADDRESS_REG_FLOAT1;
  if (memory_bool1_out[0] === `FALSE) begin
    write_enable_r  = `TRUE;
    ray_state_nxt   = RAYSTATE_ITERATIVE_31;
  end
  else begin
    ray_state_nxt   = RAYSTATE_ITERATIVE_PUSH_1;
  end
end
```

Figure 19: How it should have been done. Every signal has a default value.

As we can see in the three tables above (result from Xilinx), the maximum clock frequency decrease by 17% when two pixel processors were used, instead of only one pixel processor. The frequency decrease that much because there are longer paths because there is not optimal routing environment when one more pixel processor is added. These two processors are completely parallel, so they do not need to get longer paths if they have their own separate and equal size area on the FPGA. Florplanning can be done to make the pixel processors less dependent on each other, instead of mixing all their logic and routing together.

6 FPGA vs. CPU performance

The C++ ray tracer model that is running on a CPU achieved 2.56 FPS (section 3.11) where the Verilog ray tracer model intended for the Virtex6 FPGA achieved 0.39 FPS (section 5.2) when two pixel processors are used. The FPGA is in this case 85% slower than the CPU. This is not an impressive result, but the design changes that are mentioned in section 5.2 should decrease the longest path, decrease the area consumed by routing, and increase the number of pixel processors that can fit on the FPGA. This will then increase the FPS that is achievable.

7 Conclusions

A simple and customizable ray tracer in Verilog has been made. This ray tracer is capable of simulating optical effects such as reflection of light, and diffuse and specular light. The ray tracer has also functionality that support other optical effects. Tools that simplifie the design process and automates the verification process has been made. The FPGA ray tracer run at 0.39 frames per second. This is not satisfactory since the CPU ray tracer is running at 2.56 FPS but this can be overcome as described in section 6.

7.1 Future work

What can be done next is to

- Make reading from registers synchronously instead of asynchronously.
- Remove some of the consecutive combinatorial logic in the pixel processor controller's state machine.
- Design the topmodule.
- Test the HDL code on a FPGA.
- Add some other optical effects.
- Make the scene dynamic.

8 References

[1] IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, IEEE Std 754-2008, 2008.

[2] Lafortune, Willems, *Rendering Participating Media with Bidirectional Path Tracing*, Katholieke Universiteit Leuven, Belgium, 1996.

[3] Application programming interface by Qt, `http://qt.nokia.com/`

[4] *Virtex-6 Family Overview*, Preliminary Product Specification, Xilinx, 2011.