הפקולטה להנדסה

VLSI LAB

Hardware Implementation of Ray Tracing Algorithm

Aviner Fishhof

Ori Millo

Academic Guide: **Prof. Shmuel Wimer**

Guide: **Mr. Moshe Doron**

October 2011, תשרי תשע"יב

Table of Contents

Figures

Tables

1. Introduction

The hardware implementation of the Ray Tracing Algorithm is a way of creating 2D pictures of 3D graphical environments that look more realistic than pictures that are produced using the Rasterization method.

The algorithm is based on the idea that the way we see the world around us depends entirely on the direction, color and amount of light in it. In other words, if there is absolute dark around us we won't see a thing regardless to what objects are in front of us.

In order to see how light influence the scene in front of us we should trace each ray that is coming out of each light source in our scene. This, of course, would require a tremendous amount of work and would a great waste since most of these rays are not hitting the view plane which we are looking through. Therefore the algorithm actually goes the other way around and traces rays from the plane view to the light source while calculating on the way the influence each hit point has on the color of the pixel the ray was shot from.

The software implementation of the algorithm is widely used in the entertainment and video games industries but the problem is that even this way the algorithm still takes a lot of processing power in order to process a single frame. For example, if we assume that the processing time for a frame takes 30 seconds (process time can take much longer depends on the complexity of the scene), it will take 45 days to process an hour and a half full length featured movie.

The presented hardware implementation should accelerate the algorithm performance so image processing could occur in real-time.

2. Project Targets and Problem at Hand

The goal of this project was to develop and implement a hardware core that can perform ray tracing operations defined for handling 3D graphic images.

Computer graphics subsystems today are based on very fast multiple compute cores, having common memory pool executing the rendering operations in floating point.

This architecture is power-hungry, requires cooling systems and suited for wall-plugged Devices. There is a lack of low power; battery operated graphical cores that can serve Mobile Game Consoles, Personal Digital Assistants and Smart-Phones.

This project is a good start for a solution to the "mobile graphics" syndrome. Continuation along this track will eventually provide a quality solution to the problem.

3. Theoretical Background

3.1. Ray tracing

One way to classify rendering algorithms is according to the type of light interactions they capture.

Ray tracing describes a method for producing visual images constructed in 3D computer graphics environments, with more photorealism. This is done by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects.
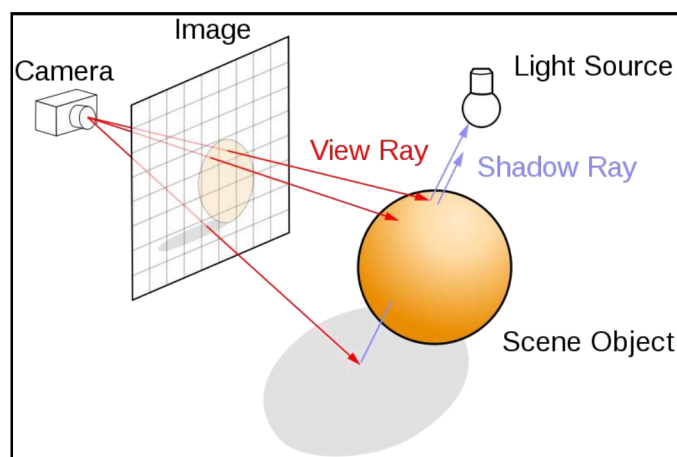
Figure 1: Ray Tracing Overview

In Figure 1 we can see a 3D computer graphics environment that consists of a single sphere and a single light source. Setting the camera (viewer) position and the image plane (aspect ratio) enables us to configure which parts of this 3D environment will take place in our final 2D image, and in which resolution.

Next, a ray is being casted from each pixel in the image plane towards the scene, and is tested for intersection with some subset of all the objects in the scene. Once the nearest object has been identified, the algorithm will estimate the incoming light at the point of intersection, examine the material properties of the object, and combine this information to calculate the final color of the pixel.
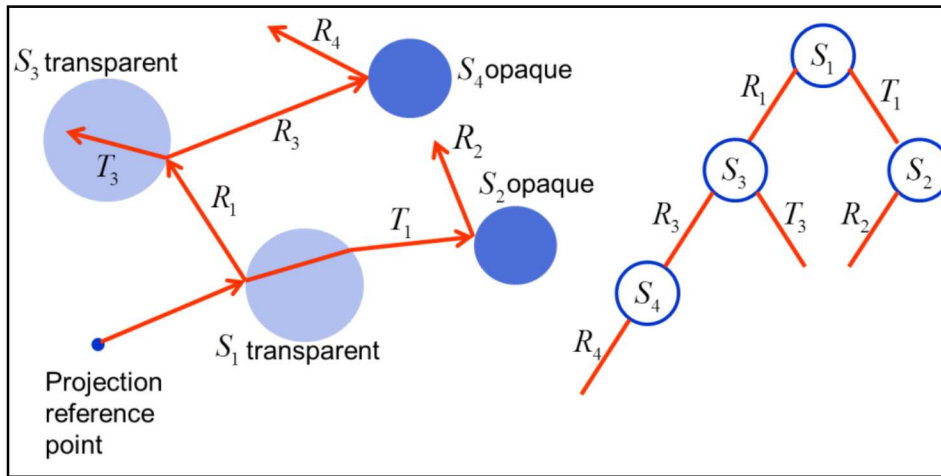
Figure 2: Binary Ray-Tracing Tree

Reflective or translucent materials may require more rays to be re-cast into the scene, thus creating a binary ray-tracing tree as seen in Figure 2. After either a maximum number of reflections or a ray traveling a certain distance without intersection, the ray ceases to travel and the pixel's value is updated.

At this point, a complete picture has been created.
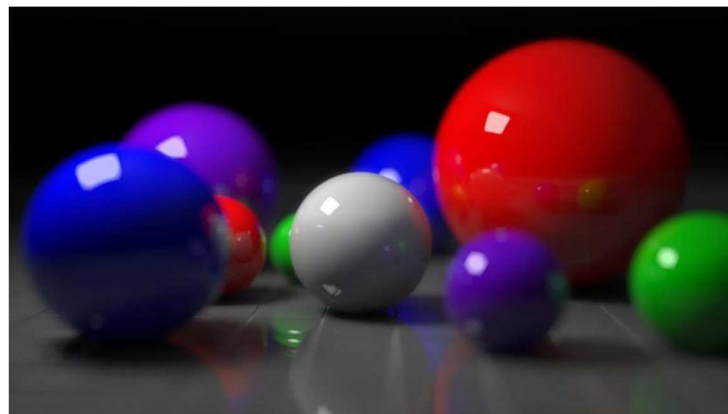
Figure 3: Ray Traced 3D Image

To determine the way rays behave when intersecting with different materials several physical models are being used:

1. Snell's Law for ray refraction.

For transparent objects, the **Snell's Law** is used to describe the relationship between the angles of incidence and refraction.

Snell's law states that the ratio of the sine's of the angles of incidence and refraction is equivalent to the ratio of phase velocities in the two media, or equivalent to the opposite ratio of the indices of refraction: $\frac{sin\,\theta_1}{sin\,\theta_1} = \frac{n_2}{n_1}$ with each θ as the angle measured from the normal and *n* as the refractive index (which is unit less) of the respective medium[1].
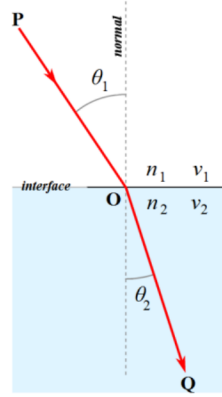
Figure 4: Snell's Law

2. Phong shading/Phong reflection model for ray reflection.

For objects that reflect the ray, using **Phong reflection model** which describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces and also includes an ambient term to account for the small amount of light that is scattered about the entire scene[2].

Each material in the scene has the following definitions:

k$_s$: specular reflection constant, the ratio of reflection of the specular term of incoming light.

k$_d$: diffuse reflection constant, the ratio of reflection of the diffuse term of incoming light (Lambertian reflectance).

k$_a$: ambient reflection constant, the ratio of reflection of the ambient term present in all points in the scene rendered.

α: shininess constant for this material, which is larger for surfaces that are smoother and mirror-like. When this constant is large the specular highlight is small.

Define lights as the set of all light sources, *L* as the direction vector from the point on the surface toward each light source, *N* as the normal at this point on the surface, *R* as the direction that a perfectly reflected ray of light would take from this point on the surface, and *V* as the viewer's direction pointing towards the scene.

$$I_p = k_a i_a + \sum_{m\,\in\,\text{lights}} \left(k_d (L_m \cdot N) i_{m,d} + k_s (R_m \cdot V)^\alpha i_{m,s} \right).$$

Equation 1: Phong equation

[1] http://en.wikipedia.org/wiki/Snell%27s_law
[2] http://en.wikipedia.org/wiki/Phong_shading

Phong equation is used for computing the shading value of each surface point while transforming it to this system's rays' direction (coming from the viewer point of view and not the other way around).

V is the viewer P.o.V and R_m is the direction vector bouncing from the surface, and is calculated by: $R_m = V - 2(N \cdot V)N$
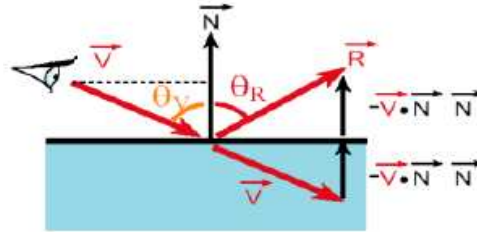
Figure 5: Phong reflection model

3. Lambertian reflectance for ray reflection and refraction.

For objects that reflect the Ray in the scene we use the Lambertian reflectance to give us the ratio of the Ray's reflection with the surface, or basically, the intensity of the Ray that moves on from the point of intersection. We calculate the dot product of the surface's normal vector, N, and a normalized light-direction vector, L, pointing from the surface to the light source[3].

3.2. RGB Color Model

Color representation is done by the commonly used RGB color model which is an additive color model in which red, green, and blue light is added together in various ways to reproduce a broad array of colors. The main purpose of the RGB color model is for the sensing, representation, and display of images in electronic systems, such as televisions and computers, though it has also been used in conventional photography. Before the electronic age, the RGB color model already had a solid theory behind it, based in human perception of colors.

To form a color with RGB, three colored light beams (one red, one green, and one blue) must be superimposed (for example by emission from a black screen, or by reflection from a white screen). Each of the three beams is called a component of that color, and each of them can have an arbitrary intensity, from fully off to fully on, in the mixture. The RGB color model is additive in the sense that the three light beams are added together, and their light spectra add, wavelength for wavelength, to make the final color's spectrum[4].

---

[3] http://en.wikipedia.org/wiki/Lambertian_reflectance
[4] http://en.wikipedia.org/wiki/RGB_color_model

4. Development Environment (Software & Hardware)

4.1. **Microsoft Visual Studio** was used to write the software implementation the Ray Tracing Algorithm in C++.

4.2. **Microsoft Office Visio Tool** was used to draw the schematics and block diagrams of the Ray Tracing Algorithm high and low level designs.

4.3. **Verilog Hardware Description Language** (HDL) was used for hardware low level design.

4.4. Both **Cadence Incisive** and **Xilinx ISE Design Suite 13.2** were used for simulation and logical verification of the design.

4.5. **Xilinx ML605 Base Board** was used to migrate the design onto an FPGA (Field Programmable Gate Array) Board, perform HW Debug Sessions and as the target hardware platform for implementing the Ray Tracing Core in a programmable (FPGA) device.

5. Design Flow

- Studied Ray Tracing algorithm throughout its software implementation in C++.

- Configured Requirements for the Ray-Tracing Core, consisting of the graphical input/output data formats and the mathematical equations of the graphical operations needed.

- Wrote the Ray Tracing Core Specification.

- Performed High-Level Design (Architecture/Block Diagrams & Execution Flow Diagrams).

- Performed Low-Level Design, employing the Verilog HDL.

- Simulated and verified logical design, using Cadence Incisive & Mentor Graphics Simulators.

- Mapped the design onto the Xilinx ML605 Development FPGA board.

- Performed Hardware debug on the FPGA Board using Board's debug.

- Ran the graphical Demo and presented results on a VGA Monitor.

6. System Software Implementation

The Ray Tracing Algorithm is a complicated algorithm that uses many arithmetic operations.

In order to understand the Ray Tracing algorithm to its full, and to get a "feel" of how it should behave we implemented the algorithm in software using C++.

This stage was essential and gave us the starting point for the hardware implementation in the following terms:

1. How the input data should look like. This includes the definition of our world coordinates, the definition of objects and definition of lights.
2. How rays are to be defined.
3. Mathematical calculations made. As seen in the previous chapters, the Ray Tracing algorithm is based on trigonometry to find the direction of each ray being casted and 2nd degree equations to find the intersection points with objects.
4. The depth of algorithm recursion. This can have a major effect on the accuracy of the final image, and also on the time a single frame is to be produced.
5. A definition of infinity value that will not harm the accuracy of the algorithm.
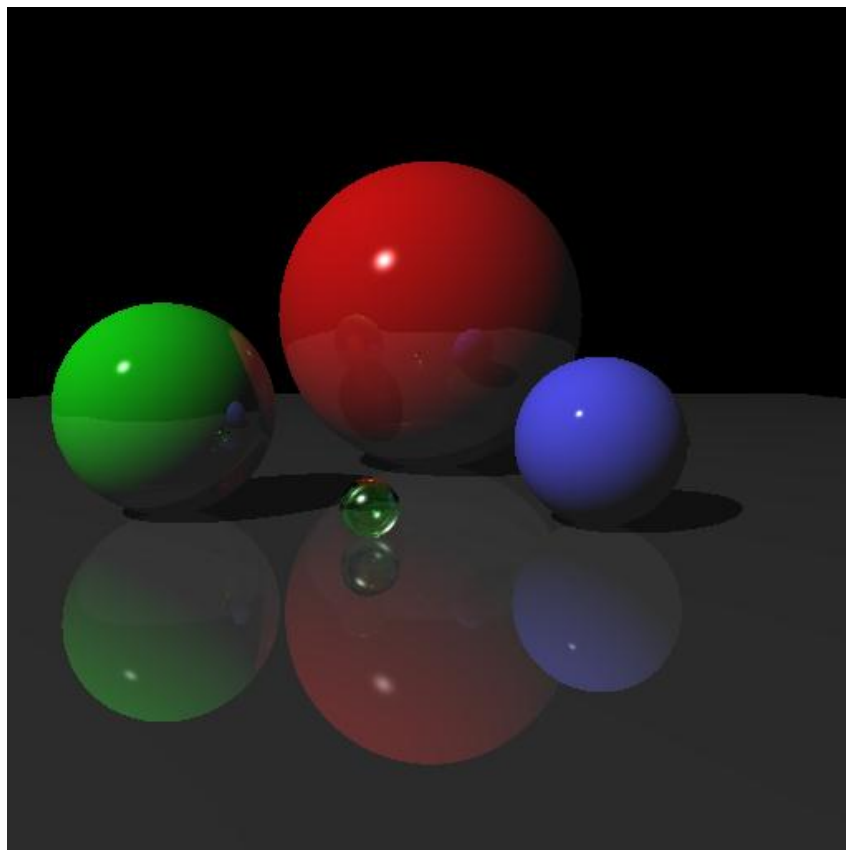
Figure 6: Ray Traced image produced by the SW implementation

7. System Block Diagram

Figure 7 describes the Ray Tracing memories pipe-line stages and the connection between them.
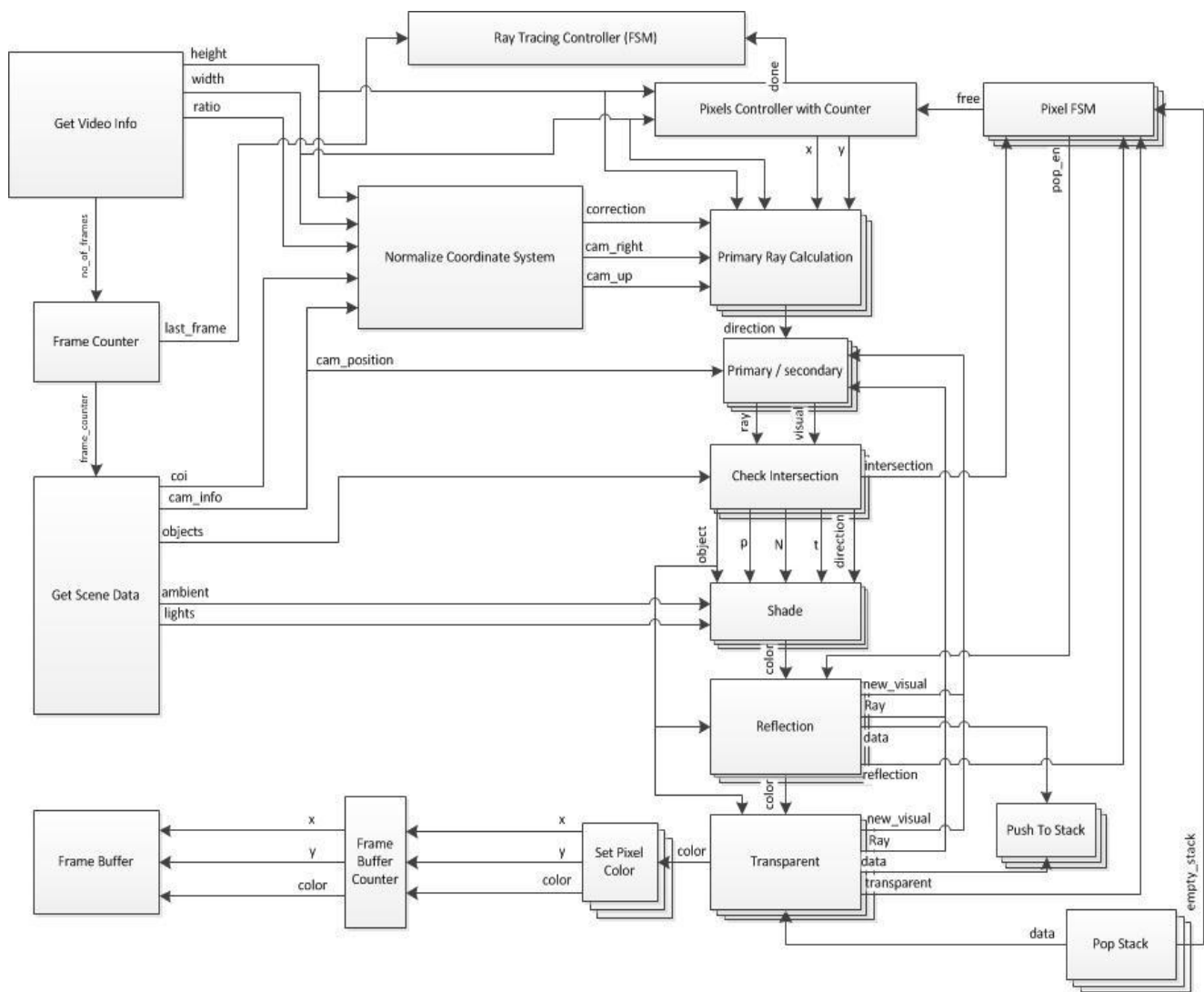
Figure 7: Ray Tracing Block Diagram

Data Types & Formats

Data	Element	Type	Size	Description
Frame	Camera position	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	The point in space representing the Point of View.
	Camera upper side direction	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	A vector representing the direction of the camera's upwards direction.
	Center of interest	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	A point in space representing the center point of the frame being ray traced.
	Background Color	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color parts combining the frame background color.

Table1 : Frame Data Types

Data	Element	Type	Size	Description
Lights	Light position	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	The point in space representing the position of the light source.
	Light intensity	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the light source.
	Ambient Light[5]	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the ambient light of the frame.

Table2 : Lights Data Types

---

[5] Ambient Light (aka available light) refers to any source of light that is not explicitly supplied to the scene.

Data	Element	Type	Size	Description
Objects (General)	Ambient color[6]	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the ambient color of the object.
	Diffuse color[7]	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the diffuse color resulted from ray hitting an object.
	Specular Color[8]	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the specular color resulted from ray hitting an object.
	Reflection Color	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the reflection color resulted from a color returned by a reflected ray casted from the point of intersection with an object.
	Transparent color	Unsigned Fixed Point [9.10]	3 * 19 bits = 57 bits	A color element representing each of the RGB color intensity combining the transparent color resulted from a color returned by a refracted ray casted from the point of intersection with an object.
	Shining constant	Integer	4 bits	A constant defining the object material in terms of smoothness.

Table3 : Object Data Source

[6] Ambient color is the color of an object when it is in shadow. This color is what the object reflects when illuminated by ambient light rather than direct light.

[7] Diffuse reflection is the reflection of light from a surface such that an incident ray is reflected at many angles.

[8] Specular reflection is the mirror-like reflection of light from a surface, in which light from a single incoming direction (a ray) is reflected into a single outgoing direction.

Data	Element	Type	Size	Description
Sphere	Center position	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	The point in space representing the sphere center position.
	Radius	Unsigned Fixed Point [2.3]	5 bits	The radius length. Can take values from 0.125 to 3.875.

Table4 : Sphere Data Types

Data	Element	Type	Size	Description
Triangle (Polygon)	Vertex 1 position	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	The point in space representing the position of first vertex constructing the triangle.
	Vertex 2 position	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	The point in space representing the position of second vertex constructing the triangle.
	Vertex 3 position	Singed Fixed Point [9.10]	3 * 19 bits = 57 bits	The point in space representing the position of third vertex constructing the triangle.

Table 5 : Triangle Data Types

7.1.1. **Frame Representation**

Each frame will include the following elements in the following order:

- Camera position - consists of x, y and z components (57 bit).
- Camera upper side direction - consists of x, y and z components (57 bit).
- Center of interest - consists of x, y and z components (57 bit).
- Number of lights (2 bits).
- Number of spheres (2 bits).
- Number of polygons (4 bits).
- Light sources (104 each).
- Spheres (352 bits each).
- Triangles (461 bits each).
- Ambient light - consists of RGB values (57 bit).
- Background color - consists of RGB values (57 bit).

7.1.2. **Light Sources Representation**

Each light source is represented by:

- Light Position - consists of x, y and z components (57 bit).
- Light Intensity - consists of RGB values (57 bit).

7.1.3. **Object Representation**

Each object will have the following color definitions representation:

- Ambient - consists of RGB values (57 bits).
- Diffuse - consists of RGB values (57 bits).
- Specular – consists of RGB values (57 bits).
- Shining – a constant value = 16, built into the system.
- Reflection - consists of RGB values (57 bits).
- Transparency - consists of RGB values (57 bits).
- Index – a decimal number between 0-15 (4 bits).

Overall object's color definitions representation requires 289 bits.

There are two kinds of objects: Spheres and Polygons. An object of each kind will be represented in a way which differs from the other by an identification bit and the physical location definitions:

- Sphere – Will be identified by a '0' in the MSB. The following 289 bits will be the color definitions as described above. The next 57 bits are the sphere's center coordinates followed by its radius (5 bits).

0	1 — 57	58 — 114	115 — 171	172 — 228
0	Diffuse	Specular	Ambient	Reflection

229 — 285	286 — 289	290 — 346	347 — 351
Transparency	Index	Center	Radius
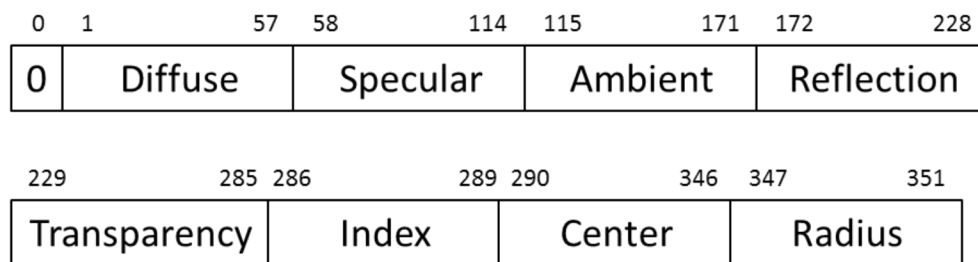
Sphere representation takes 352 bits.

- Triangle – Will be identified by a '1' in the MSB. The following 289 bits will be the color definitions as described above.
  The next 171 bits are the triangle's vertices coordinates each represented by 57 bits.

0	1 — 57	58 — 114	115 — 171	172 — 228
1	Diffuse	Specular	Ambient	Reflection

229 — 285	286 — 289	290 — 346	347 — 403	404 — 460
Transparency	Index	1st Vertex	2nd Vertex	3rd Vertex
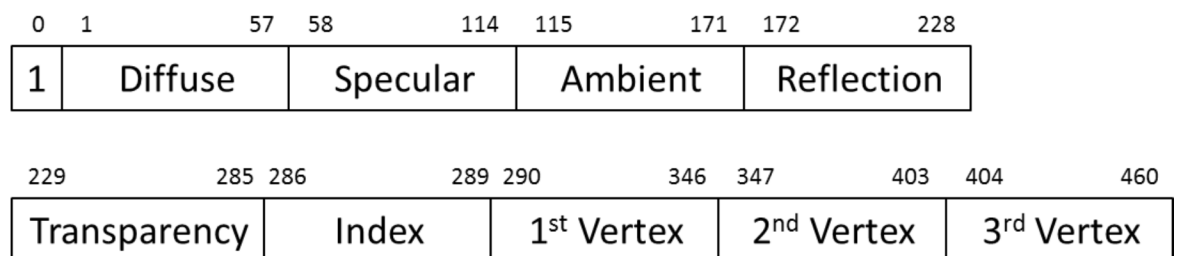
Triangle representation takes 461 bits.

8. Detailed Ray Tracing Functionality and Design

The architecture of the ray tracing design is pipelined based.
This section describes in detail every stage of the pipe.

8.1. Overall view of the design

The design flow is presented in figure 8 showing the overall view of the pipeline stages and design.

The main stages in the pipe are:

- Initialization (where reading world coordinates, screen resolution, viewer (camera) position and scene (objects data)).

- World coordinate to screen coordinate transformation.

- Checking ray's intersection with an object.

- Calculating lights' contribution to the pixel's color.

- Calculating object's reflection influence on pixel's color (if any).

- Calculating object's transparency influence on pixel's color (if any).

- Write pixel color to frame buffer.

Figure 8: Ray Tracing Global State Machine / Flow Design
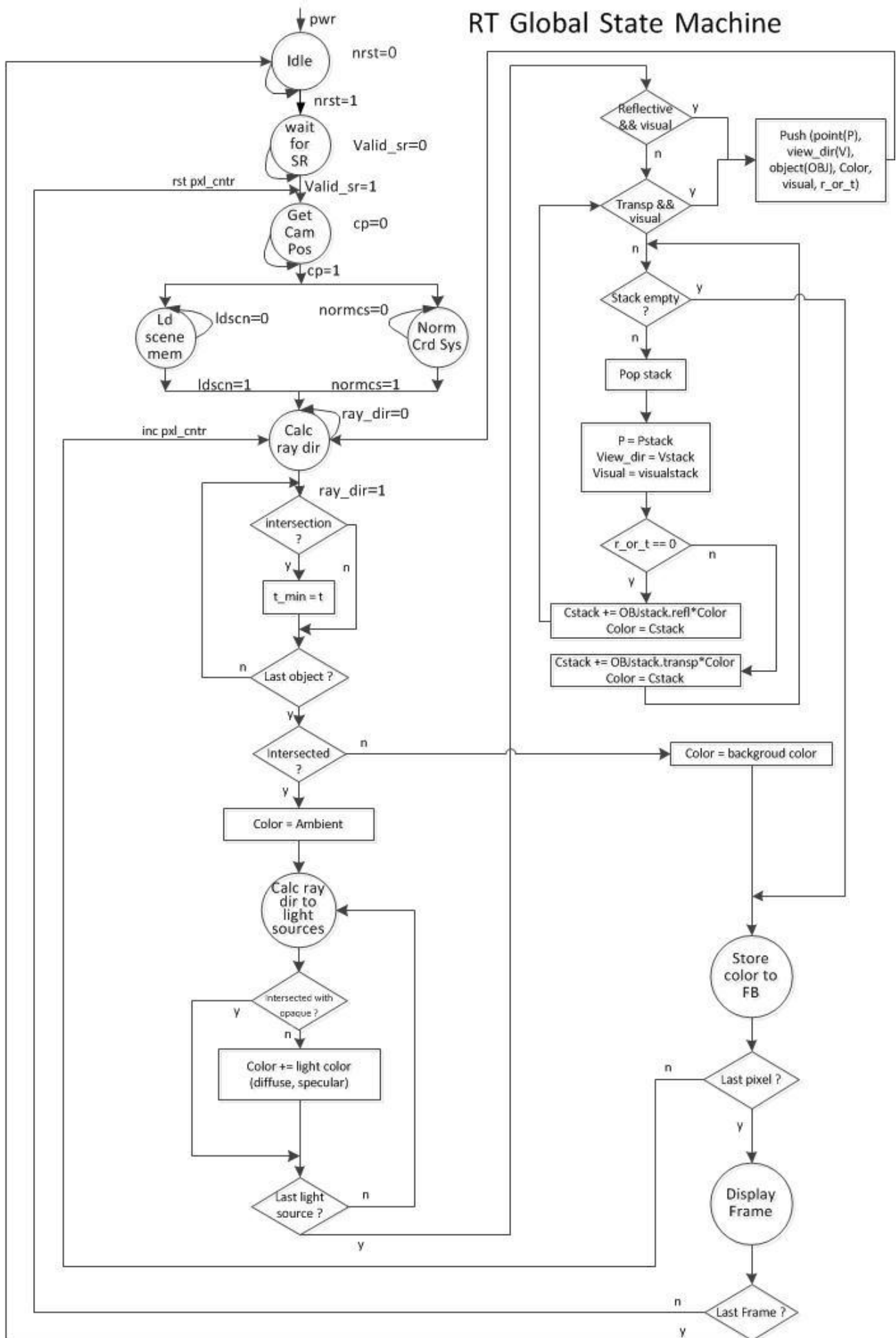
21

8.2. **Pipe Stages**

8.2.1. **Transformation and normalization of world coordinate to screen coordinate**

Genenral description:

This operation tansforms the real world coordinates to a screen world coordinates.

Since a world coordinate system is infinite, one must have a **point of view** in order to view it. Also, a **target point** to view at is necessary, since we don't have (yet…) the ability to see 360 degrees at a time. Last is the **normal direction** of the world were viewing (up or upsidedown?).

We define the point of view P as our "camera position", and the target point C as the "center of intrest". With these two points, we create a vector $\vec{d}$ which represents the direction that the camera is looking at.

$$\vec{d} = C - P$$

Then, using the crross product $\vec{u}$ between the normal direction $\vec{up}$ of the world and the direction were looking at $\vec{d}$, and then, the cross product $\vec{v}$ between $\vec{u}$ and $\vec{d}$ the we are able to create a view plane spreaded by $\vec{v}$ and $\vec{u}$ which is the grid of pixels to shoot rays from.

$$\vec{u} = \frac{\vec{d} \times \vec{up}}{|\vec{d} \times \vec{up}|}$$

$$\vec{v} = \frac{\vec{d} \times \vec{u}}{|\vec{d} \times \vec{u}|}$$

At this point we have the view plane of our final image, but it is still infinte. To make it finite, we must have the screen resolution of the final image, meaning the **height (h)** and **width (w)** of the frame we wish to generate, also refered to as **aspect ratio**. In our implementation we have decided to allow frames with the following aspect ratios: 1:1, 4:3 and 16:9. Changing this element between frames is not an option. Once a user choosed an aspect ratio, it will be consisntent for all frames.

An additional element is the **field of view (fov),** which is basically the zoom level of the frame. In our implementation we have decided to create all frame in a single consistent horizontal field of view **(fov_h)** zoom level, not applicable to change by the user. The vertical field of view **(fov_v)** is a fuction of the horizontal field of view and the aspect ratio.

$$fov\_h = \frac{30}{180} * \pi$$

$$fov\_v = fov\_h * \frac{h}{w}$$

To know the center position of every pixel in the view plane (spreaded by $\vec{v}$ and $\vec{u}$), we first find the *x-axis* origin:

$$x_o = \vec{u} \cdot tan(fov\_h)$$

Then the *y-axis* origin:

$$y_o = \vec{v} \cdot tan(fov_v)$$

This gives us the origin of each pixel. Now adding 0.5 to x_o and y_o will give us the center of a pixel with a small offset since all rays are being shot from a single point but each ray have a different angle hit to the view plane. At this point we add a parameter called **correction** given by $camera\_direction - camera\_right\_direction * tan(fov\_h) + camera\_up\_new * tan(fov\_v)$ that is added to the previously found direction.

From this point we can shoot a primary ray into the world!
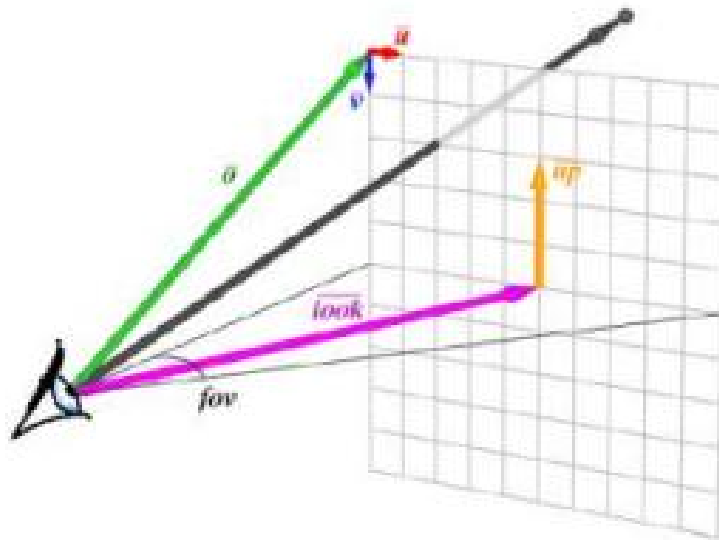
Figure 9: Constructing Primary Rays
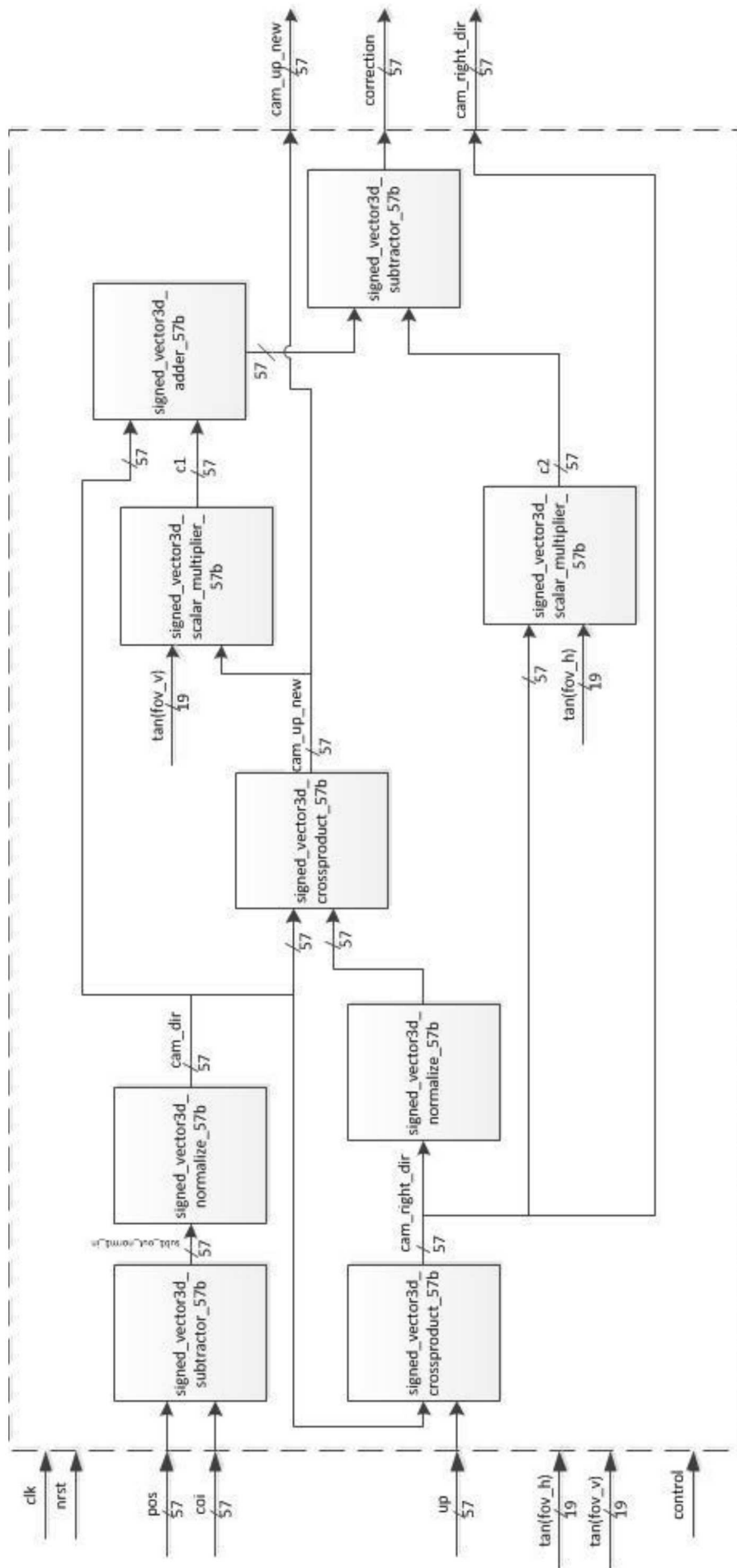
Figure10 : World Coordinate Normalize Block Diagram

8.2.2. **Intersection of Rays with Objects**

<u>General Description</u>

This operation checks for an intersection point between a Ray and each of the objects defined in the scene.

Objects in a scene may appear in every point in the world, some objects can be closer to the view plane than others and some can hide others.

Rays are vectors, starting at an origin point and are infinite straight lines. These infinite lines can intersect multiple objects in their path.

In our implementation there are two types of objects that can be produced: polygons, which are consrtucted by triangles, and spheres.

<u>Ray-Plane intersection</u>

Ray R is a line:

$$(i) \qquad x(t) = x_0 + td$$

Polygon is constructed by triangles, each triangle T lies on a plane. To get the intersection of R and T, one first determines the intersection of R and the plane:

$$(ii) \qquad n \cdot x = 0$$

Placing (i) in (ii):

$$=> \qquad n \cdot (x_0 + td) = 0$$

$$=> \qquad (n \cdot d)t + (n \cdot x_0) = 0$$

$$=> \qquad t = \frac{-(n \cdot x_0)}{(n \cdot d)}$$

To intersect a ray with a 3d triangle which is denoted by its three vertices first compute the triangle edge vectors:

$$u = p_1 - p_0$$
$$v = p_2 - p_0$$

25

Compute the cross product between to get triangle normal. If the outcome of the cross product is degenerate it means that the triangle is a segment or a point, and the algorithm does not deal with this case.

Next, according to the direction of the ray, and the dot product of the noraml with a vector from the ray origin to a triangle vertex, we can tell if the ray is parallel to the triangle, lies on it or disjoint from it.
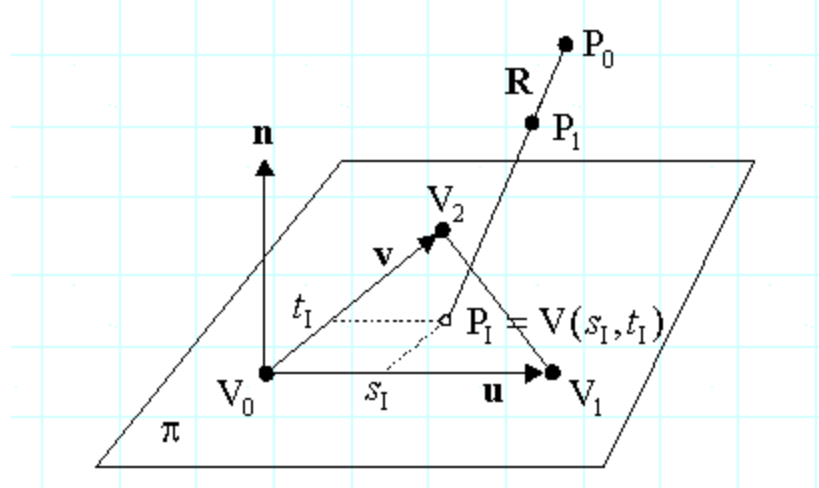
Figure 11: Ray Triangle Intersection

When passing the phases above, check to see if the ray direction is towards the triangle, if so find the intersection point using:

$$s_I = \frac{\mathbf{w} \cdot \mathbf{v}^\perp}{\mathbf{u} \cdot \mathbf{v}^\perp} = \frac{\mathbf{w} \cdot (\mathbf{n} \times \mathbf{v})}{\mathbf{u} \cdot (\mathbf{n} \times \mathbf{v})} \quad \text{and} \quad t_I = \frac{\mathbf{w} \cdot \mathbf{u}^\perp}{\mathbf{v} \cdot \mathbf{u}^\perp} = \frac{\mathbf{w} \cdot (\mathbf{n} \times \mathbf{u})}{\mathbf{v} \cdot (\mathbf{n} \times \mathbf{u})}$$

Figure 12 is the flow that describes checking an intersection of a ray with a triangle.
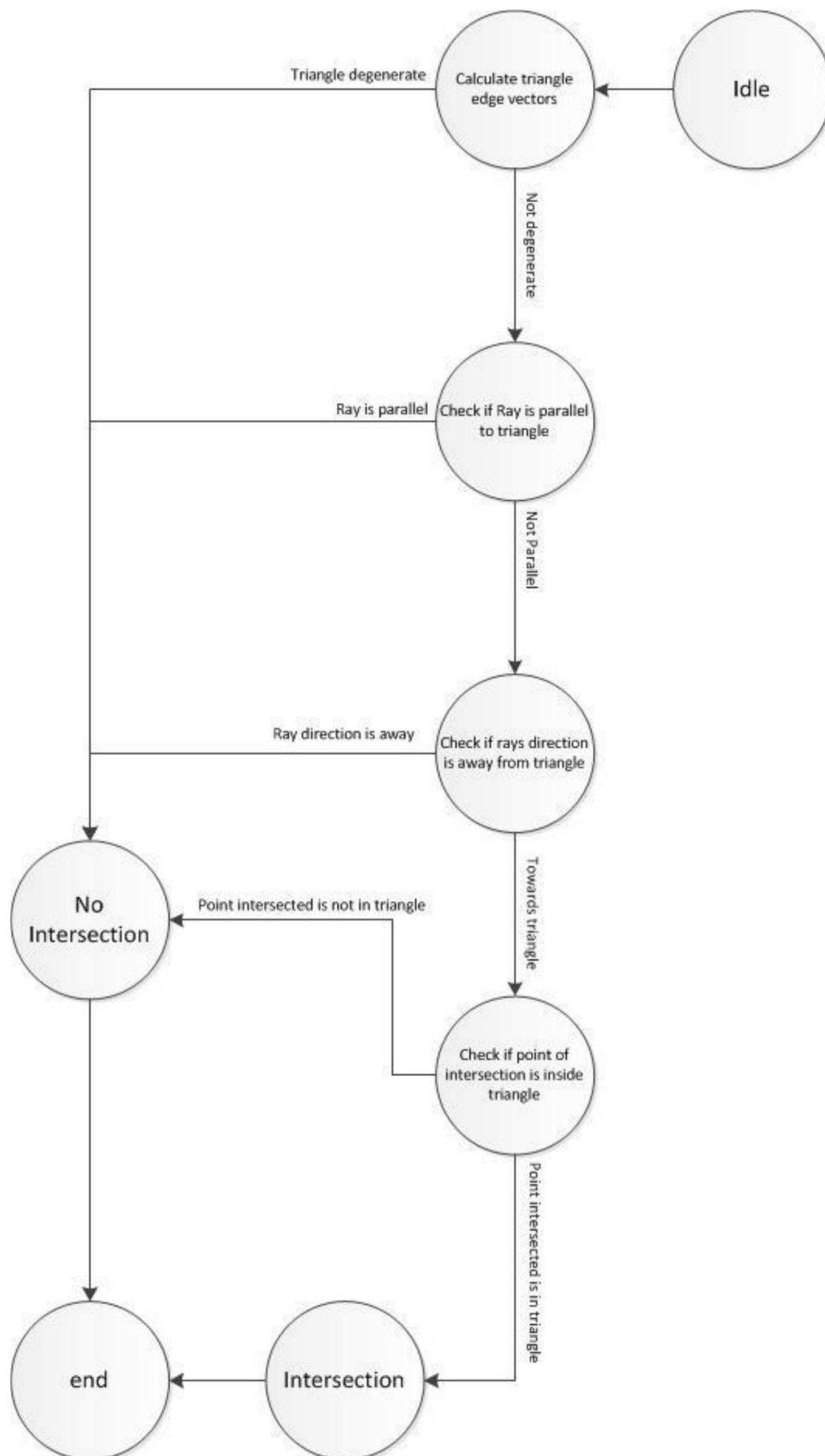
Figure12 : Ray Triangle Intersection Flow

Ray-Sphere intersection

A line and a sphere can intersect in three ways: no intersection at all, at exactly one point, or in two points.

A sphere is given by

$$\|x - c\|^2 = r^2$$

Where c is the center point of the sphere and r is the radius.

A line is given by

$$x = dI$$

Where d is the distance along line from starting point and I is the direction (a unit vector).

Solving for D

$$\|dI - c\|^2 = r^2$$

$$\Rightarrow \qquad d = (I \cdot c) \pm \sqrt{(I \cdot c)^2 - c^2 + r^2}$$

If the value under the square root is less than zero, then it is clear that no solution exist and the line doesn't intersect the sphere.

If it is zero, then exactly one solution exists and the line touches the sphere in one point.

If it is greater than zero, two solutions exist and the line touches the sphere in two points.
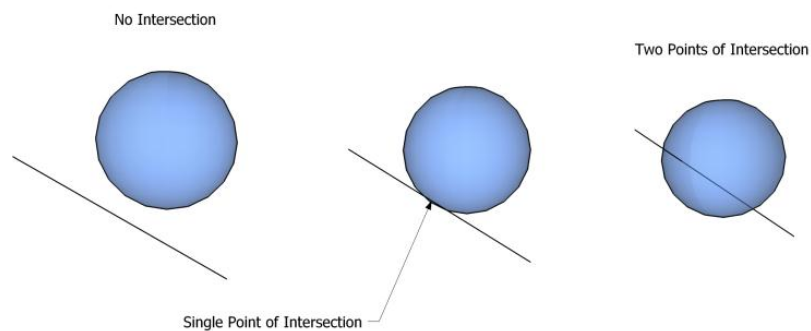
Figure 13: Ray-Sphere Intersection

Check Intersection Block Diagram

Init_sub		
dot_products1		
dot_products2		
AC_calc		
muls		
discriminant_calc		
discriminant_sqrt		
ts_calc		
pick_t		
P_calc1		
P_calc2		
N_calc		

Figure14 : Ray-Sphere Intersection Block Diagram

29

8.2.3. **Lights Contribution to Points of Intersection (Shade)**

<u>General description</u>

This operation checks if lights in the scene contribute direct light color to point of intersection with object, or is the intersection point is shaded.

Once a point of intersection is reached, the algorithm calculates the color in that intersection point to add for the final pixel color from which the primary ray was released.

The color in that intersection point is calculated by the color of the object the ray intersected with, the color of secondary ray to be released from the intersection point (reflective and transparent ray to be discussed in next section) and the color contributed by light sources in the scene or shades resulted from these light sources or any other objects in the scene.

In the point of intersection, according to the Phong reflection model shown in figure 15, a new ray is created for each of the light sources, with the intersection point as the origin of the ray and the light source as the ray direction.

For each one of these rays, the algorithm checks if there is an object between the light source and the point of intersection. If there is no object, then the light source contributes it color directly onto the point of intersection. In case an object (could be more than one) is in the way, check its material (transparency) to see if the light source contributes through the object in the way or not at all, in which case, the point of intersection is shaded.
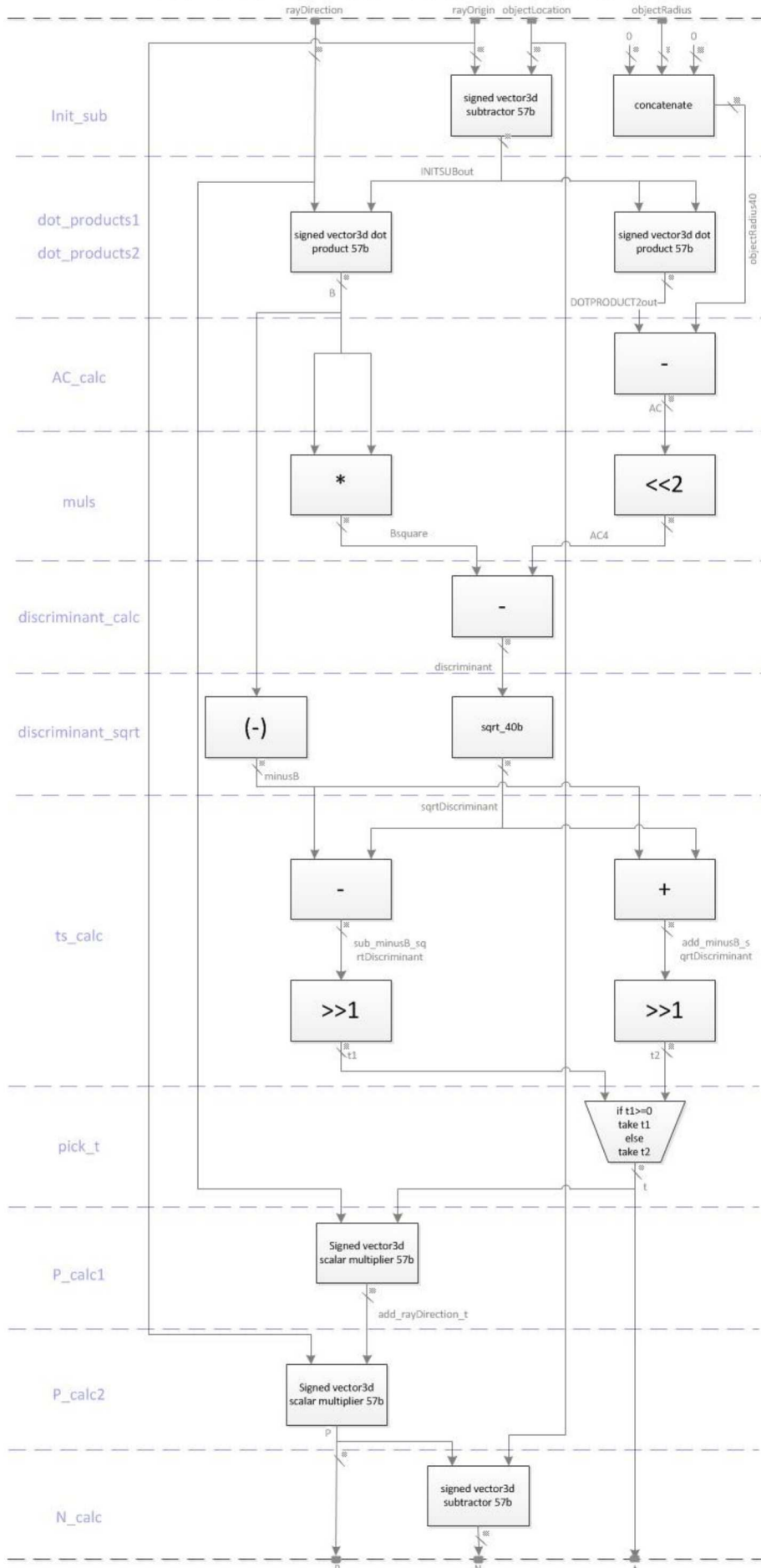
<u>Calculations made in this stage</u>

The first state in this stage is combines:

- Finding the *L* vector which is the ray from the intersection point to a light source.
  This is done by knowing the position of a light source, and subtracting the point of intersection from it to get a vector directed from the intersection point towards the light source. Once we have that vector, normalize it by dividing it with its norm.
- First color calculations. Whether the intersection point is in the shade or not, it still receives the scene ambient light which is then multiplied by the object's ambient to receive the first color contribution for the pixel at hand.

The second state of this stage combines:

- Checking an intersection of the new light ray *L* with other objects in the scene. This to see if the intersection point needs to be added a diffuse color element and a specular color element. If there is an object intersected with *L* we move on to create new reflected and transparent rays from intersection point.

- To make good use of our resources, while waiting for answer if *L* intersected or not, we start to perform preparations to later on compute diffuse and specular elements. This to save time if *L* doesn't intersect with any object.

The third state of this stage sums all color elements calculated in this stage to a color register of 57 bits.

Figure 16 shows the block diagram of this stage.
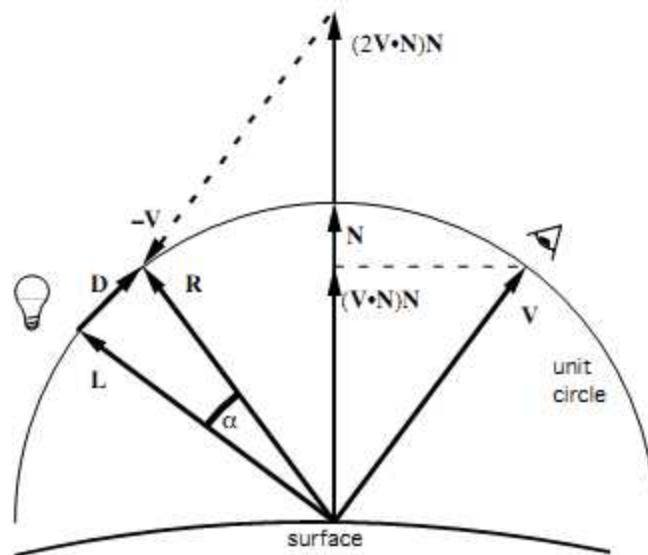
Figure 15: Phong Reflection Model with a Light Element

31

Figure 16: Shade Stage Block Diagram

32

8.2.4. **Reflective Rays Calculations**

General description

This operation checks if the object a ray intersected with has reflective values. If so, generate a reflective ray.

Using Phong equation mentioned in chapter 3 (Theoretical Background), the algorithm constructs a new reflective ray with the intersection point as its origin and R as its direction, where R is:

$$R = V - 2(N \cdot V)N$$

V is the incoming ray direction; N is the intersection point normal.

The reflected ray created is casted as a secondary ray into the scene to find the next intersection point that contributes to the final color of the pixel.

If this ray intersects another object, the same intersection process begins for that intersection point.

When do we stop?

The first answer to that is when the reflective ray doesn't intersect any object and the color returned from that ray is the background color.

The second option is to accumulate product of reflection coefficients and stop when this product is too small.

The third option is to stop at a fixed depth.

8.2.5. **Transparent Rays Calculation**

General description

This operation checks if the object a ray intersected with has transparent values. If so, generate a transparent ray.

Using Snell's law mentioned in chapter 3 (Theoretical Background), the algorithm constructs a new transparent ray with the intersection point as its origin and T as its direction.

Calculation of T:

$$Snell's\ Law:\qquad \frac{\sin\theta_1}{\sin\theta_1} = \frac{n_2}{n_1} = \eta$$

$$T = M\sin\theta_t - N\cos\theta_t$$

$$M = \frac{(N\cos\theta_i - I)}{\sin\theta_i}$$

$$T = \frac{\sin\theta_t}{\sin\theta_i}(N\cos\theta_i - I) - N\cos\theta_t$$

$$T = N\,(\eta\cos\theta_i - \cos\theta_t) - \eta I$$

$$\cos\theta_i = N \cdot I$$

$$\cos\theta_t = \sqrt{1 - \sin^2\theta_t} = \sqrt{1 - \eta^2(1 - (N \cdot I)^2)}$$

$$T = \left(\eta(N \cdot I) - \sqrt{1 - \eta^2(1 - (N \cdot I)^2)}\right)N - \eta I$$
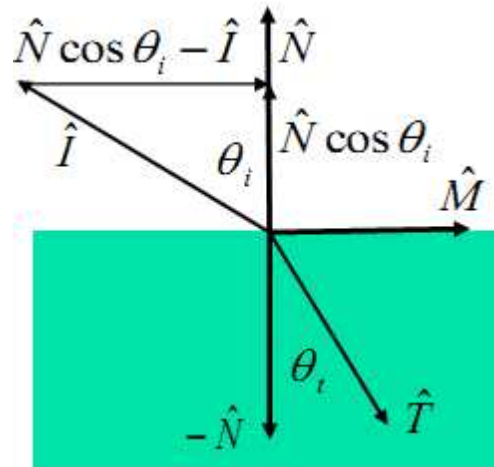
Figure17 : Transparent Ray

8.2.6. **Writing Pixel Color to Frame Buffer**

After a primary ray was casted from a center of a pixel, its color is calculated throughout all previous stages stated above.

In case the ray hits an object, the first parameter, the ambient element, of the color is being calculated.

Then, depending on light sources and other objects in the scene, we add shading elements to the color, diffuse and specular.

Depending on the object intersected, the color of the pixel is being added a reflection element and a transparent element. Both of these elements are ready with color values of their own.

When the procedure ends, the color element which is an *RGB* values in 19*3 bits is multiplied by 255 and adjusted to fit an 8 bit *RGB* color model that can later be an output to a VGA screen.

Multiplying by 255 is because the color values being calculated throughout the algorithm are supposed to be in the range of [0,1], whereas the *RGB* color model should receive values in the range [0,255].

The *RGB* values obtained for the above calculation are written into a Frame Buffer that is eventually sent to the VGA port for display on Host's screen. This takes place at the rate of 24 frames per second. It is filled pixel by pixel as a result of the above color calculation. Once filled, the Frame Buffer is flushed to the VGA port.

9. Ray Tracing Design Considerations

Implementing the ray tracing algorithm in software was used to understand the algorithm to its full, but was also used as a helpful tool of reference.

For example, when a primary ray is casted, its initial intersection point is set to infinity, which in the software implementation is a matter of assigning INF to a variable. But when building the hardware design it suddenly became clear that not only INF must have a real value now but it also has to be the minimal value that will be as accurate as possible on one hand but still humble in the resources it demands on the other. In this case, committing a reasonable number of simulations using the software implementation, showed the point where an object is positioned but is not visible and that point's value was set to be infinity.

Considerations made

1. Defining the size of our world.
   Through our software simulations we came up that when an object is placed at a distance of 250 from the center of interest it is no longer viewable. Therefor our world can be considered as a sphere with a radius of 250.

2. Defining precision.
   Hardware requires a fixed point values representation. Since the algorithm values are very often within the range of [1,-1] we had to decide on the precision that will make it possible to implement the algorithm without affecting the final result, but still to take into account that our resources are limited.
   We came to the decision that a 10 bit precision, that gives us 1/1024 accuracy level, after the decimal point will be sufficient. This decision was also made after using the software implementation as the simulator and printing out to screen a fairly large number of calculations results.

3. Timing considerations.
   The main reason for implementing ray tracing in hardware is to achieve real time video playing without the need for rendering. Again, using the software implementation, we measured the time it took to produce a single frame in various situations and combinations of the world producing the frame to. For example, creating a 512 pixels height by 512 pixels width frame for a world

with 3 spheres, 2 triangles (forming a square) and 3 light sources took 30 seconds. When narrowing the components in the world to just one sphere and on light source it took 5 seconds. When taking these results into account, and that out resources is limited, we had to minimize the number of objects we are able to include in the world, the number of light sources and since the ray tracing algorithm is recursive, to minimize the depth of the recursion, all this to achieve real time frame producing under the time and resources limits we had.

10. Ray Tracing Hardware Implementation

Implementing the algorithm in hardware consisted of the following:

1. Simulating modules.

   In order to verify our modules, a test bench was written for each module.

2. Synthesize overall design.

   The simulation tool we used and the

10.1. Implementation of Arithmetic modules

The ray tracing design consists of many arithmetic modules that were needed to be implemented specifically to match the algorithm requirements.

10.1.1. Adder

The main data types used in the algorithm are vectors that represent our rays. Each of these vectors has three parts x, y and z.

$$\bar{V} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Summing two vectors require summing each part of the vectors individually:

$$\bar{V_1} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \quad ; \quad \bar{V_2} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \quad ; \quad \bar{V_3} = \begin{pmatrix} x_3 = x_1 + x_2 \\ y_3 = y_1 + y_2 \\ z_3 = z_1 + z_2 \end{pmatrix}$$

When performing these operations in bits, there is an option of overflow. So, when V_1 is 57 bits long, where each part is separately 19 bits, summing x_1 with x_2 will give us a 20 bit result that is needed to be truncated back to 19 bits with a value that will not cause problems with precision in the next level.

The 19 bit representation of a vector part was designed with the careful thought of the values the algorithm is in need of, so when two parts are summed, and the result has an overflow,

we assign the result with the default maximum or minimum (when summing two negative numbers) value our system can deal with. When the summing action doesn't create an overflow, the 19 LSB's holds the required result.

Example:

$$\overline{V_1} = \begin{pmatrix} 1.5 \\ 1.5 \\ 1.5 \end{pmatrix} \; ; \; \overline{V_2} = \begin{pmatrix} 3.25 \\ 3.25 \\ 3.25 \end{pmatrix} \; ; \; \overline{V_3} = \begin{pmatrix} 4.75 \\ 4.75 \\ 4.75 \end{pmatrix}$$

$$\overline{V_1} = \overbrace{[000000001.1000000000]}^{x} \overbrace{[000000001.1000000000]}^{y} \overbrace{[000000001.1000000000]}^{z}$$

$$\overline{V_2} = \overbrace{[000000011.0100000000]}^{x} \overbrace{[000000011.0100000000]}^{y} \overbrace{[000000011.0100000000]}^{z}$$

$$\begin{aligned} & 000000001.1000000000 \\ + \; & 000000011.0100000000 \\ \hline & 000000100.1100000000 \end{aligned}$$

10.1.2. Subtraction

The concept of subtraction is the same as addition described above with the only difference of subtracting instead of adding:

$$\overline{V_1} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \; ; \; \overline{V_2} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \; ; \; \overline{V_3} = \begin{pmatrix} x_3 = x_1 - x_2 \\ y_3 = y_1 - y_2 \\ z_3 = z_1 - z_2 \end{pmatrix}$$

10.1.3. Multiplication

10.1.3.1. Vector and Scalar Multiplication

This module receives a vector in the representation mentioned above and a scalar which is represented in 19 bits, in the same way as each one of the vector elements.

Multiplying 19bit number with another 19 bit number outputs a 38 bit result. Since the result is still a vector in the system, we need it in 19 bits, therefor truncating the result. We know that the number of bits after the decimal point was 10, so now it is 20, and that the number of bits before the decimal point was 9, so now its 18. Truncating the 10 LSB's and the 9 MSB's will give us the desired result, this after checking the result didn't exceed the maximum or minimum values the system is designed for.

Example:

$$\bar{V} = \begin{pmatrix} 1.5 \\ 1.5 \\ 1.5 \end{pmatrix} \quad ; \quad s = 3.25 \quad ; \quad s\bar{V} = \begin{pmatrix} 1.5 * 3.25 = 4.875 \\ 1.5 * 3.25 = 4.875 \\ 1.5 * 3.25 = 4.875 \end{pmatrix}$$

$$\bar{V} = \overbrace{[000000001.1000000000]}^{x} \overbrace{[000000001.1000000000]}^{y} \overbrace{[000000001.1000000000]}^{z}$$

$$s = [000000011.0100000000]$$

$$
\begin{array}{r}
000000001.1000000000 \\
\times \ 000000011.0100000000 \\
\hline
00000000000000000000 \\
00000000000000000000 \\
000000000.0000000000 \\
\hline
\end{array}
$$

0000000000000000100.11100000000000000000

10.1.3.2. Point Wise Vector Multiplication

This module receives two vectors and multiplies each part of the first vector with its corresponding part in the second vector.

$$\overline{V_1} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \quad ; \quad \overline{V_2} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \quad ; \quad \overline{V_3} = \begin{pmatrix} x_3 = x_1 * x_2 \\ y_3 = y_1 * y_2 \\ z_3 = z_1 * z_2 \end{pmatrix}$$

The calculation made is practically the same as done in the previous vector and scalar multiplication.

10.1.4. Division

There are a couple routines that require division in the ray tracing algorithm. The first is to normalize a vector, dividing each of its parts with its norm. The second is when calculating the 2nd degree equation to find an intersection point of a ray with a sphere.

Keeping the precision along through all phases of the algorithm requires us to deal with small numbers, with values that are less than 1. We had to find a way that can be used to divide fractional numbers that are less than the value 1, with numbers that are smaller than themselves, without getting 0 as a result. At the same time, we needed it to be able also dealing with any other value necessary, and most important, not damaging the precision.

The solution to this was to shift left dividend by 10 bits, perform the division, and take the 19 LSB's as the result, where 10 LSB's are the fractional part.

When simulating this we found out that when shifting the dividend by more bits the result is much more accurate.

Example:
1. Divide 2 (2'b10) by 3 (2'b11), with 6 binary fractional resolution (right from the decimal point).

2. Shift (2'b10) left by 8.
   10 << 2;   result 1000000000.

3. Perform division:

```
    10101010
1000000000 |11
   11
    10
    100
    11
    10
    100
    11
     10
     100
     11
      10
```

4. "Shift" result right by 8.
   .10101010

5. Result = 1/2 + 1/8 + 1/32 + 1/128 = 0.6640625 (0.39% inaccuracy)

Adding additional resolution bit, will yield (+1/512) 0.666015625 (0.065% inaccuracy)
The division operation takes 2 clock cycles; the first is to compute the division result for each part of the vector simultaneously and the second to output the result.

10.1.5. <u>Vector operations</u>

As described above, the ray tracing algorithm requires various vector operations.

10.1.5.1.   Dot Product
Taking two vectors (of equal length) and returning a single number obtained by multiplying corresponding entries and then summing those products.
The dot product of vectors expressed in an orthonormal basis is related to their length and angle. For such two vectors V and U, the dot product is given by

$$\bar{V} \cdot \bar{U} = \|V\| \|U\| \cos \theta$$

where θ is the angle between them
This operation is useful of calculating the direction of new rays, light rays, reflective rays and refraction rays at intersection points.

The dot product operation takes 2 clock cycles; the first is to compute the multiplication result for each part of the vector simultaneously and the second to sum all three results and output the final result.

10.1.5.2. Cross Product

Is an operation on two vectors in three-dimensional space. It results in a vector which is perpendicular to both of the vectors being multiplied and normal to the plane containing them.
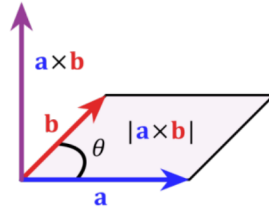

Figure 18: Cross Product

The cross product is given by:

$$\overline{V_1} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \quad ; \quad \overline{V_2} = \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} \quad ; \quad \overline{V_1} \times \overline{V_2} = \overline{V_3} = \begin{pmatrix} y_1 z_2 - z_1 y_2 \\ z_1 x_2 - x_1 z_2 \\ x_1 y_2 - y_1 x_2 \end{pmatrix}$$

This operation is in use when normalizing the world's coordinate system and enables us to define vectors that are orthonormal base to a view plane which is used as the grid of pixels from which the primary rays are being casted.

10.1.5.3. Norm

Norm is a function that assigns a strictly positive length to vectors in a vector space. Defined by:

$$\overline{V_1} = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \quad ; \quad \|\overline{V_1}\| = \sqrt{(x_1 * x_1) + (y_1 * y_1) + (z_1 * z_1)}$$

This operation is one of the most significant in the ray tracing algorithm. Since most vector related calculations require the use of normalized vectors, finding the norm of a vector become an essential operation that cannot be avoided.

10.1.6. Square Root

The main issue in finding a vector's norm is calculating the square root of the sum of its point wise multiplications.

Square root is also the main issue in finding the intersection point of a ray with a sphere (solving 2nd degree equation $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$).

These two issues are the most common operations used in the ray tracing algorithm. Every ray is a vector that starts as normalized vector, and every ray has the chance to intersect with a sphere object.

Being so commonly used, we needed the calculation of the square root to efficient. There are a few algorithms dealing with the subject. At first our thought was to use the 'Newton-Raphson', but although it if relatively fast (can take two iterations) when using a ROM table for storing a set of initial guesses, it uses a lot of hardware resources and the final result is yet just an approximation. As mentioned previously, precision is one of the most important issues in the ray tracing algorithm, and for this reason and the fact that we need as much more resources we can get in order to maximize the parallelism of our system using 'Newton-Raphson' wasn't the optimal solution.

Our decision was to use the 'Non-Restoring' algorithm described in "A New Non-Restoring Square Root Algorithm and Its VLSI Implementations" by Yamin Li and Wanming Chu. This algorithm requires only one traditional adder/subtractor in each iteration, i.e., it does not require other hardware components, such as seed generators, multipliers, or even multiplexor. It generates the correct resulting value even in the last bit position. Next, based on the resulting value of the last bit, a precise remainder can be obtained immediately without any correction or addition operation. Finally, it can be implemented at very fast clock rate because of the very simple operations at each iteration. Figure 19 show a pseudo code of the non-restoring algorithm.

```
unsigned squart(D, r) /*Non-Restoring sqrt*/
unsigned D;        /*D:32-bit unsigned integer
                      to be square rooted */
int     *r;
{
  unsigned Q=0;  /*Q:16-bit unsigned integer
                                    (root)*/
  int R=0;  /*R:17-bit integer (remainder)*/

  int i;

  for (i=15;i>=0;i--)  /*for each root bit*/
  {
     if (R>=0)
     {                        /*new remainder:*/
       R=(R<<2)|((D>>(i+i))&3);
       R=R-((Q<<2)|1);             /*-Q01*/
     }
     else
     {                        /*new remainder:*/
       R=(R<<2)|((D>>(i+i))&3);
       R=R+((Q<<2)|3);             /*+Q11*/
     }
     if (R>=0) Q=(Q<<1)|1;       /*new Q:*/
     else      Q=(Q<<1)|0;       /*new Q:*/
  }


                    /*remainder adjusting*/
  if (R<0) R= R+((Q<<1)|1);
  *r=R;                   /*return remainder*/
  return(Q);                  /*return root*/
}
```

Figure 19: Non-Restoring Algorithm Pseudo Code

In our ray tracing system, the non-restoring algorithm is applied on a D = 40 bit [20.20] number, therefor it takes D/2 clock cycles to complete.

11. Encountered Problems and Solutions

During the course of this project we've encountered the following problems:

11.1. All vectors must start as a unit magnitude when being used during intersection calculations. Failing to do so can corrupt the entire algorithm behavior.
Although the operation of normalizing a vector is the most time consuming operation in our design we have decided that this will not be considered as a limitation.

11.2. Divisions – Ray Tracing require divisions mainly to normalize vectors. At first our thought was to use a ROM that will save us time and resources. When progressing through the implementation and simulations, it turned out that although we save time and resources, we pay for that in precision. The main guideline in our system was for it to be as precise as possible, and that made us to change the design in this part and make the system division friendly.

11.3. Our software implementation was a complicated ray tracer that could handle various environments containing number of objects and light sources when each object could have almost any texture combining different amounts of transparency, reflection and shininess.
Along the progress of the project's design and implementation we came to acknowledge that implementing that ray tracer in hardware might be impossible for us in the given time frame because of two main reasons:

i. The Ray Tracing Algorithm is a recursive algorithm which is rather complicated to implement in hardware. Doing so requires using a stack and some extensive additional control system on top of the one that already exist.

ii. A hardware system that is designed to work with unknown number of objects (in our system his could mean either objects presented in the world or light sources) requires special control systems or loop circles which are more complicated to design and construct.

Therefore, the project's objectives were changed as follows:

> Displaying only one object – Doing so allows us to avoid the search of the nearest object to the point view or light source and save these calculations and in addition it makes it unnecessary to calculate secondary rays that are reflected from the object since where ever the ray goes it will hit nothing.

> The object is a sphere – When using a pre-determined figure for the object there is no need to implement the parts in the algorithm that relate to other figures (triangles in this matter).

> One light source - Makes it easier to find the shaded parts of the object.

> No transparency – Allows skipping the calculation of secondary rays that are created as a

result of passing through a transparent material.

Changing the project's objectives this way allowed the avoidance of designing and building the extra control parts.

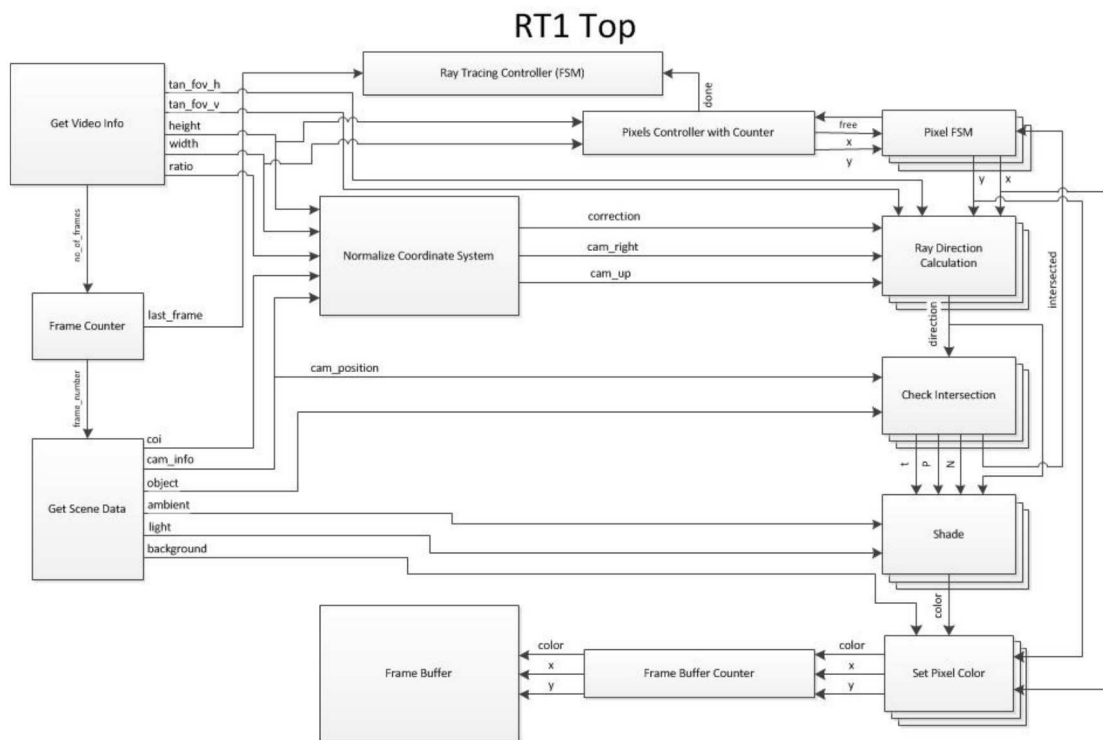The new design is as presented below:

Figure 20: Final design block diagram

12. Conclusions and Summary

Devoting considerable time and effort to understand the mathematical requirements for the Ray Tracing algorithm through software, allowed us to better understand the implications of Hardware implementation, with its associated limitations.

Adhering to the Design Documentation requirements (Spec, Hi-Level Design, Flow Diagrams, etc.) enabled us to incorporate the entire, detailed design intricacies into the Project Book .
This will serve as a good baseline for continuation projects like Ray Tracing Backend and designing a Ray Tracing USB device.

13. Ideas for Continuation

Since the final implementation was just a part of the complete design, the next phase can be implementing the overall design with our implementation, which was implemented with careful thought for being easy to use in the future without the need of remodeling, as the base design or continuing the development of our implementation to a full USB device.

The purpose of this project was to implement a simple ray tracer in hardware, and so is done. There are methods to accelerate the ray tracing algorithm such as to enclose groups of adjacent objects in spheres, calculate intersection with enclosing sphere first, and proceed to real objects only if intersection exists, or Space-Subdivision, where the entire scene is enclosed with a cube which is then divided into smaller cubes until the number of spaces contained in a cube doesn't exceed a predefined limit or the size of cube reached some threshold. Taking our implementation and accelerate it with these techniques can increase the ability of the ray tracer future USB device.

14. Bibliographic List

14.1. **Computer Graphics Lecture Notes by Prof. Wimer**.

http://www.eng.biu.ac.il/~wimers/Courses/Computer_Graphics/Lecture_Notes

14.2. **Ray Tracing from the Ground Up, Kevin Suffern. Chapter 3.**

14.3. **The OpenGL SuperBible, 5th edition, Richard S. Wright, Nicholas Haemel , Graham Sellers , Benjamin Lipchak .**

14.4. **http://www.asic-world.com/verilog/intro1.html**

14.5. **"A New Non-Restoring Square Root Algorithm and Its VLSI Implementations" by Yamin Li and Wanming Chu, Computer Architecture Laboratory The University of Aizu, Japan**

14.6. **Phong Shading Reformulation for Hardware Renderer Simplification, Apple Technical Report #43, Richard F. Lyon.**