Ray-Cast Three-Dimensional Pong

Elizabeth Power!
Richard Hughes

December 2008

Abstract

Project Name: Ray-Cast Three-Dimensional Pong
Project Team Members: Elizabeth Power!, Richard Hughes

In this project we created a 3-D variant of the pong project from lab 5 with ray-traced graphics. The game will have three dimensions of movement for the spherical puck and two dimensions of movement for the square paddle, and it will keep track of your score (i.e., how many consecutive bounces you've managed) on-screen with hardware 'sprite' characters. The puck will bounce off the paddle at different angles depending on the relative position of the puck to the paddle. The ray-tracing will include shadows, checkerboard walls, and 8-bit color, with 3 bit red and green and 2 bit blue.

Table of Contents

I. Abstract — Page i
II. Table of Contents — Page ii
III. List of Figures — Page iv
IV. Overview — Page 1
- Figure 0: Ray-Casting
- Figure 1: Python Generated Scene

V. Description

A. Backend — *Elizabeth Power!* — Page 2
- Figure 2: Pong Backend — Physics
- Figure 3: Physics Block Diagram

1. Ready Module — Page 4
- Figure 4: Ready Module Counting Diagram
- Figure 5: Ready Test Bench Results

2. Puck Module — Page 4
- Figure 6: Puck Module Block Diagram
- Figure 7: Puck Test Bench Results

3. Paddle Module — Page 6
- Figure 8: Paddle Module Block Diagram
- Figure 9: Paddle Test Bench Results

4. Game Module — Page 7
- Figure 10: Game Module Block Diagram
- Figure 11: Game Test Bench Results

5. Score Module — Page 10
- Figure 12: Score Module Block Diagram
- Figure 13: Score Test Bench Results

B. Graphics — *Richard Hughes* — Page 11
- Figure 14: Overall Structure of the Pipelined Ray-Caster

1. Ray-Tracing and Ray-Casting — Page 12

2. Ray-Generator Module — Page 13

3. Ray Format — Page 14
- Figure 15: Ray Generator Figure

4. Sphere-Intersector Module — Page 15
- Figure 16: Sphere Intersector Diagram

5. Plane-Intersector Module — Page 17
- Figure 17: Plane Intersector Diagram

6. Color Manager — Page 18
- Figure 18: Color Managing Diagram

VI. Conclusion Page 19

VII. References & Appendices

Appendix A: Ready Module & Test Bench Page 20

Appendix B: Puck Module & Test Bench Page 21

Appendix C: Paddle Module & Test Bench Page 24

Appendix D: Game Module & Test Bench Page 25

Appendix E: Score Module & Test Bench Page 26

Appendix F: pong_game.v Page 31

Appendix G: ray_generator.v Page 35

Appendix H: fast_sphere_intersector.v Page 36

Appendix I: plane_intersector.v Page 40

Appendix J: good_color_manager.v Page 43

List of Figures

Figure 0:	Ray-Casting	Page 1
Figure 1:	Python Generated Scene	Page 2
Figure 2:	Pong Backend — Physics	Page 3
Figure 3:	Physics Block Diagram	Page 3
Figure 4:	Ready Module Counting Diagram	Page 4
Figure 5:	Ready Test Bench Results	Page 4
Figure 6:	Puck Module Block Diagram	Page 5
Figure 7:	Puck Test Bench Results	Page 5
Figure 8:	Paddle Module Block Diagram	Page 6
Figure 9:	Paddle Test Bench Results	Page 7
Figure 10:	Game Module Block Diagram	Page 8
Figure 11:	Game Test Bench Results	Page 9
Figure 12:	Score Module Block Diagram	Page 10
Figure 13:	Score Test Bench Results	Page 10
Figure 14:	Overall Structure of the Pipelined Ray-Caster	Page 12
Figure 15:	Ray Generator Figure	Page 14
Figure 16:	Sphere Intersector Diagram	Page 16
Figure 17:	Plane Intersector Diagram	Page 18
Figure 19:	Color Managing Diagram	Page 19

Overview

Ray-tracing is a graphics rendering algorithm that logically renders a three-dimensional scene pixel-by-pixel. Each pixel corresponds to a single ray projected from a particular ‘view point’; the rays are projected as if to intersect with the ‘pixels’ on an imaginary ‘screen’ defined in the **3Dwhat?**. Geometrical analysis is performed to determine what objects in the simulated 3D space each ray intersects and we perceive the closest intersection. We then perform geometric analysis to determine how much light each light-source casts on to that intersection surface by casting rays from the intersection to the light and determining the angle of incidence, how distant and bright the light is, and whether the light is shadowed. We sum this light, determine how much of it is being sent towards the screen, and that is the color of the pixel. We then repeat that for *every* pixel.

In Figure 0, below, the ‘center of projection’ represents the ‘view point’ and the surface of the picture plane represents the screen. This illustrates how one might determine what point on the screen corresponds to a given point in 3-dimensional space.

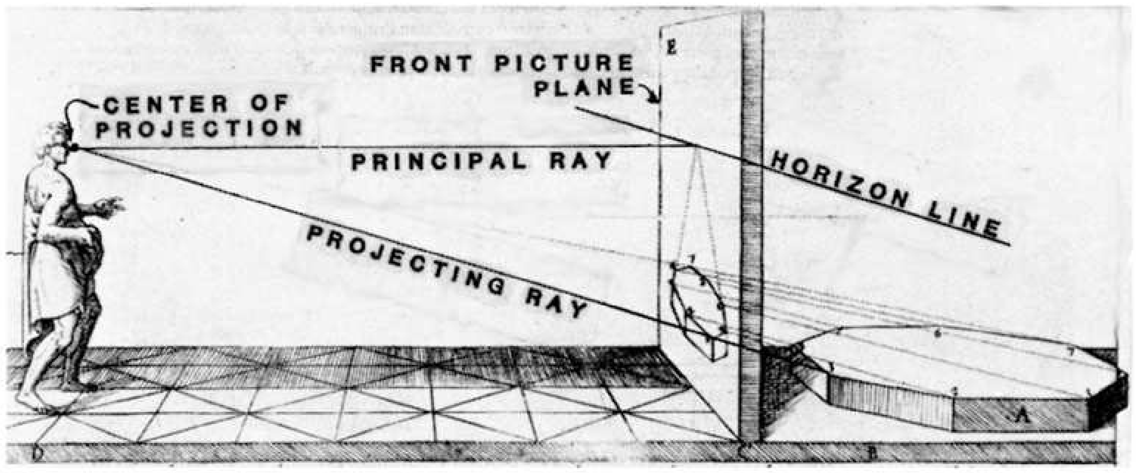


Figure 0: Ray-Casting*

Technically, this process is called ‘ray-casting’, meaning that it is not iterative — ray-tracing is, technically, an iterative form of this process that uses rays cast off from the point of intersection to accurately model such effects as reflection, transparency, refraction, or shadows that blur with distance. We will not be attempting to model such effects as reflection or refraction and so recasting is sufficient for our needs. As a result, the mathematics required to calculate the intersection of a raw with any given geometry are *bounded*, and thus we can run each intersection-calculation in parallel without worrying about one taking a far longer time than another.

Using this graphics system we are able to three dimensionally represent the physics of our Pong game. Within the game, we will use all of the basic rules of traditional Pong, with a twist. Traditional Pong is a two dimensional simulation of table tennis, where the players hit a ball back and forth with paddles and are allowed to bounce the ball off the walls. Our Pong will use a “puck,” as opposed to a ball, that will bounce off the walls in

* Image courtesy "The Arrow in the Eye" by Michael Kubovy, Christopher Tyler and WebExhibits.

the scene at the same angle which it hits the wall. There is also a paddle that will be used to keep the puck within the scene and collect points, just as in traditional Pong.
Our scene will include five planes and a sphere, as illustrated in Figure 1 below. Four of the planes — left side, right side, top, & bottom — will be sloped so that they are smaller at the back of the space and therefore appear farther away. The fifth plane will contain the controllable, square paddle and be at the back of the space. These five plans will be superimposed behind the sphere. In order to calculate where, and if, a ray intersects with the sphere, we need to calculate a square root of a fixed-point real number and perform division.

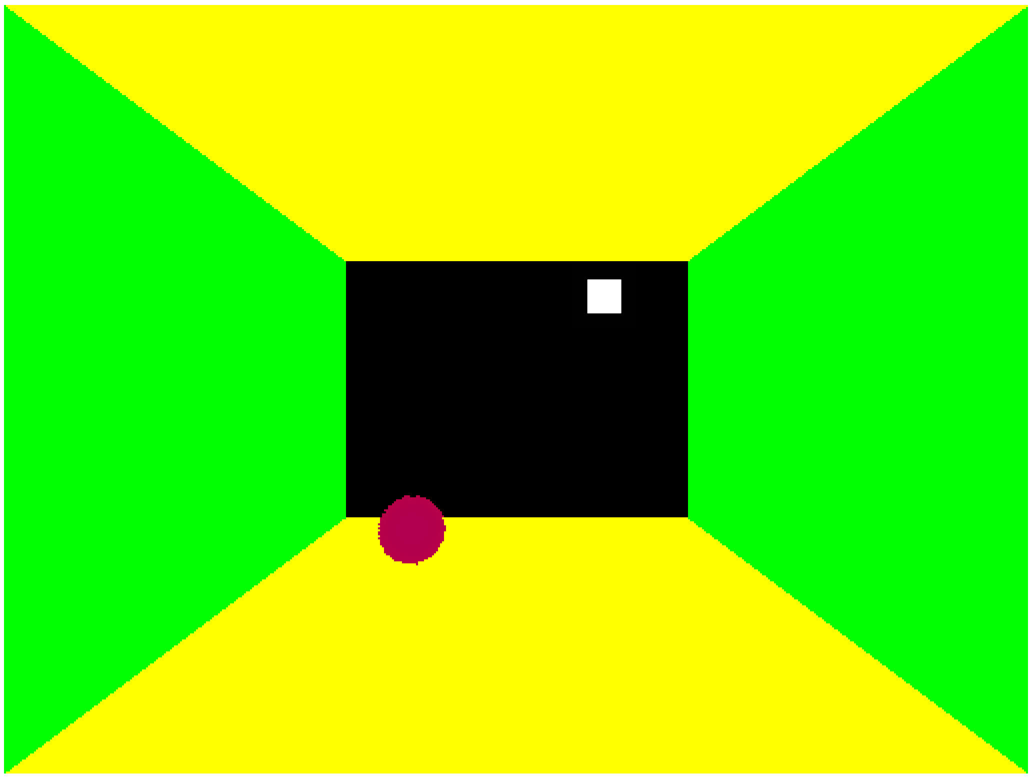


Figure 1: Python Generated Scene

The goal of the traditional game is to earn more points than the opponent, which are earned when one fails to return the ball to the other. The main difference between our three dimensional version and the traditional two dimensional version, other than the extra dimensional, is that our game is a one player version. Because of this, we have changed the goals and scoring of our players: the goal is to reach 63 points, which are earned by catching the ball with the puck.

Description — Backend

Elizabeth Power!

We have been referring to the backend of our system the “Physics” section because that is where all of the actual physical interactions take place. Our backend follows most of the laws of physics — we chose to ignore certain forces like gravity and friction — in order to help the game appear as realistic as possible.

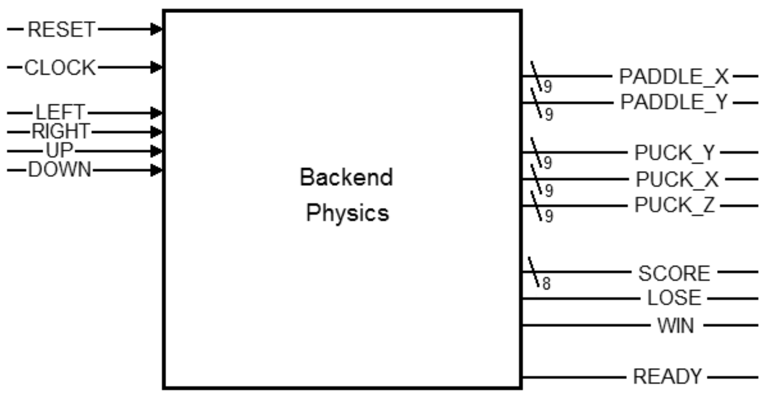


Figure 2: Pong Backend — Physics

As illustrated in Figure 2, above, the back end will use the inputs left, right, up, and down as well as loading the same Reset into each module. It also takes the clock signal and loads that into the Ready Module for the 30Hz system clock. The system will output the 9-bit X/Y/Z coordinates of the puck and paddle (Puck$_x$, Puck$_y$, Puck$_z$, Paddle$_x$, & Paddle$_y$), and the 8-bit score array, as well as single bit win, lose and ready signals. In order to simplify testability, I have also divided up the backend in to the five self-contained modules in Figure 3 (below): Ready, Puck, Paddle, Game, and Score.

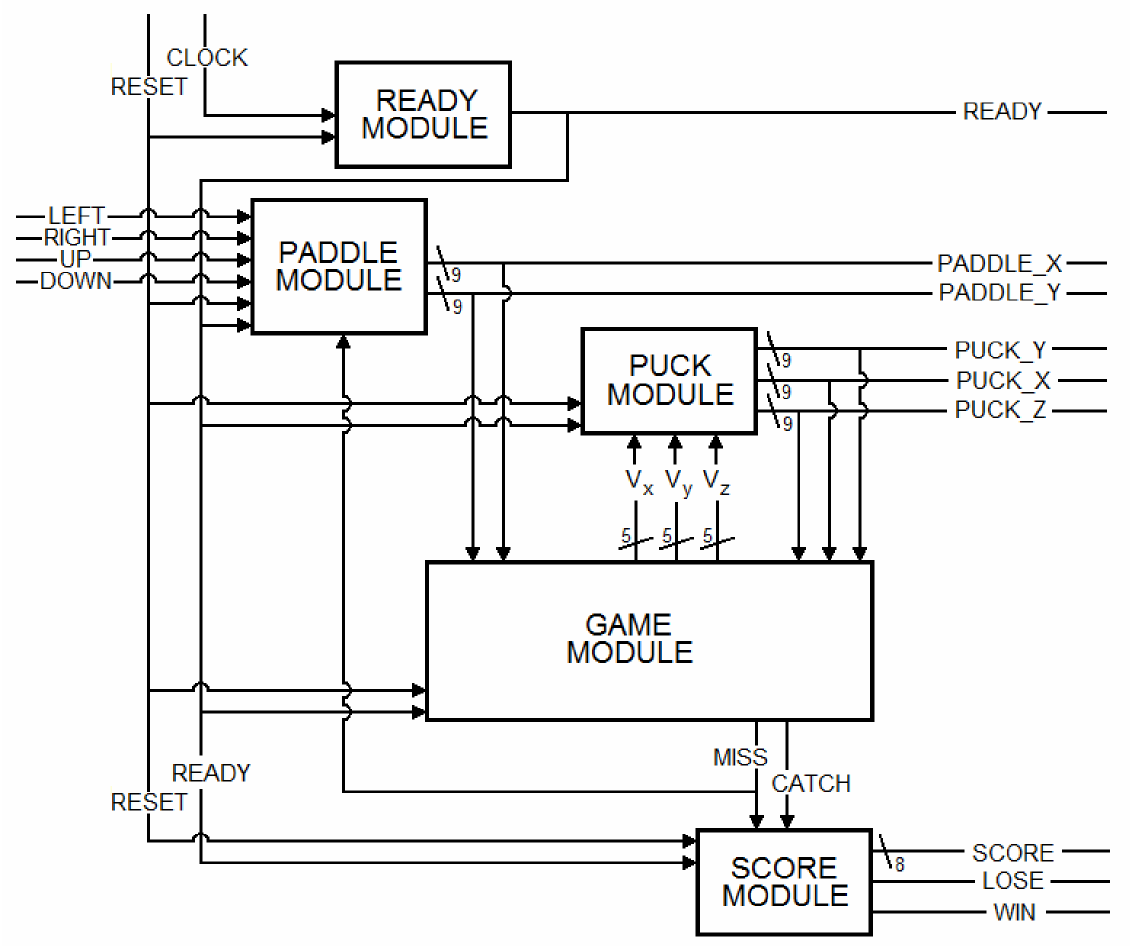


Figure 3: Physics Block Diagram

Ready Module

In order to control the game at a playable speed, the Physics Division created a “Ready” signal for the system to use. This ready signal also helps to reduce synchronization issues between the Physics & Graphic Divisions. The Ready Module works as a simple counter to convert the 27MHz clock into a 30Hz pulse signal. We will use the simple counting logic pictured in Figure 4 below: every .33 of a second (Count = 2 166 667), the signal will output a high enable for one clock cycle, otherwise it will output a low and add one to the count each 27MHz clock cycle.

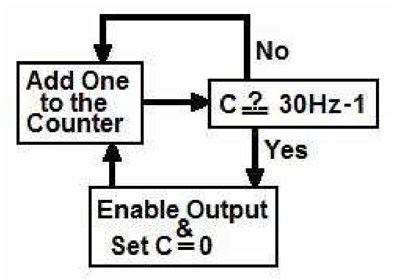


Figure 4: Ready Module Counting Diagram

As illustrated in image A of Figure 5 below, we temporarily increased the ready frequency in order to simplify testing the ready module. Examine Appendix A for the Ready Module Verilog code and test bench. Image A shows that the module increments at each positive clock edge properly and that each time Ready is enabled it is only high for one clock cycle. Image B of Figure 5 below shows the same functionality, zoomed out such that the repeating ready signal is visible.

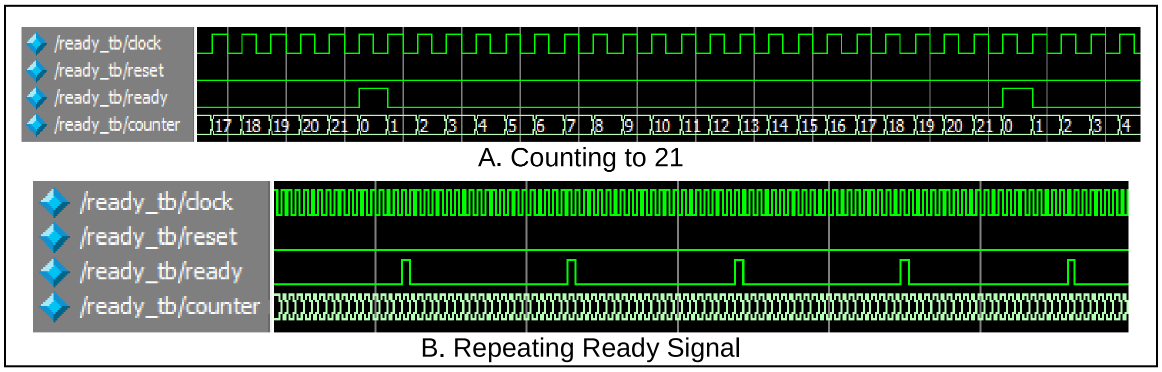


Figure 5: Ready Test Bench Results

Puck Module

The only duty of the Puck Module is to go at a designated speed and, when the player loses, stop moving. As illustrated in Figure 6 on the next page, this module takes the prerequisite Reset and Ready signals, as well as the desired 5-bit x, y, and z velocities (V_x, V_y, & V_z). The Puck Module will output the three 9-bit x, y, and z coordinates ($Puck_x$, $Puck_y$, & $Puck_z$) for the puck’s location at any given clock cycle.

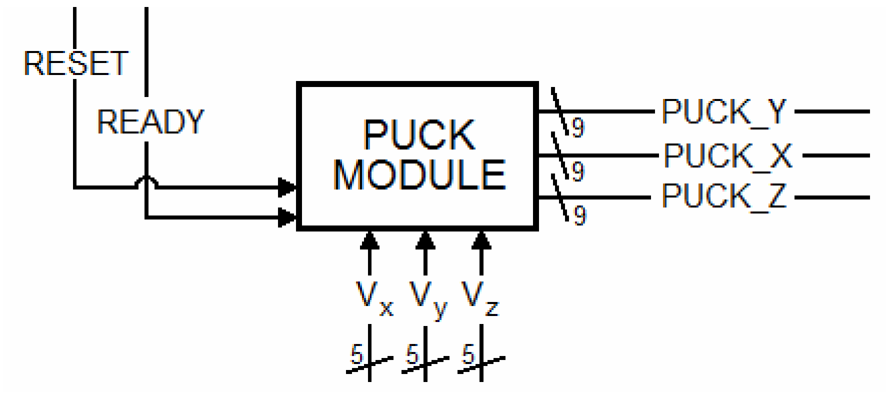
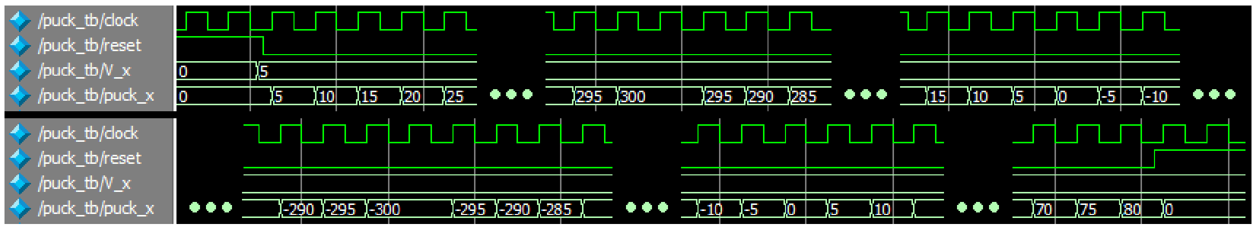


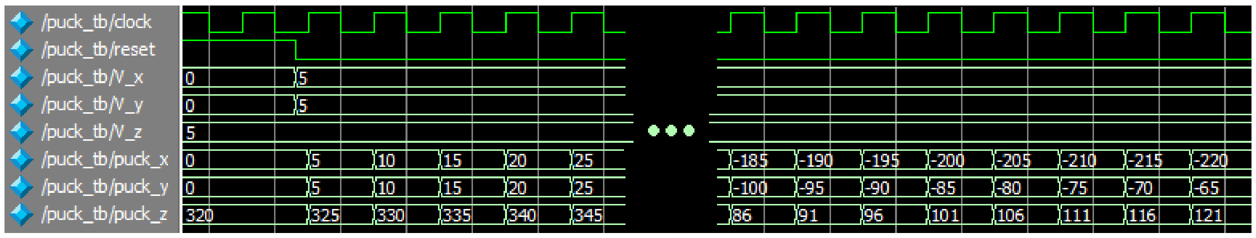
Figure 6: Puck Module Block Diagram

The function of the Puck Module is to keep track of the three-dimensional location of the puck as it travels around the scene with the variable velocities V_x, V_y, & V_z. Each axis of movement is controlled individually: at each ready cycle the $Puck_x$ will change by $|V_x|$ in the current direction (positive or negative). When the Puck "hits a wall" it will "bounce off" at the same angle. This can be accomplished by simply inverting the direction of movement for that axis — when the $Puck_x$ (moving at $+V_x$) reaches 300 (wall is at 320 & R_{Puck} = 20), it will switch to moving at $-V_x$ — irregardless of the y-axis and z-axis movement. Refer to Appendix B for the Verilog implementation of this module.

Testing for the Puck Module was fairly simple — refer to Appendix B for the Test Bench code. The first step was to test the puck's movement along each of the axes as, Image A of Figure 7 below shows selected segments of movement along the x-axis. As you can see, $Puck_x$ changes by $|V_x|$ in one direction until it reaches the coordinate of that axis' wall (in this case 300) — after that point, $Puck_x$ will change by $|V_x|$ in the other direction. Image A shows the puck easily moving though the 0 coordinate and bouncing off of both the positive and negative walls (ie right and left). After verifying that each axis works properly, Image B of Figure 7 shows that each axis functions independently of and at the same time as each other.



A. Movement on X-Axis



B. Movement on Three Axes

Figure 7: Puck Test Bench Results

As stated previously, the puck will stop moving whenever the paddle misses the puck — indicating that the game is over and the player has lost. This functionality is not implemented within the Puck Module, but within the Game Module (discussed on **page FOO**). The Game Module will change V_x, V_y, & V_z to be 0, therefore keeping $Puck_x$, $Puck_y$, and $Puck_z$ from changing and preventing the puck from moving.

Paddle Module

The only duty of the Paddle Module is to follow the input directions. As illustrated in Figure 8 below, this module takes the prerequisite Reset and Ready signals, in the internal miss signal, as well as the game control inputs (Up, Down, Left and Right). The Paddle Module will output the two 9-bit x and y coordinates ($Paddle_x$ & $Paddle_y$) for the paddle's location on the back wall of the space at any given ready cycle.

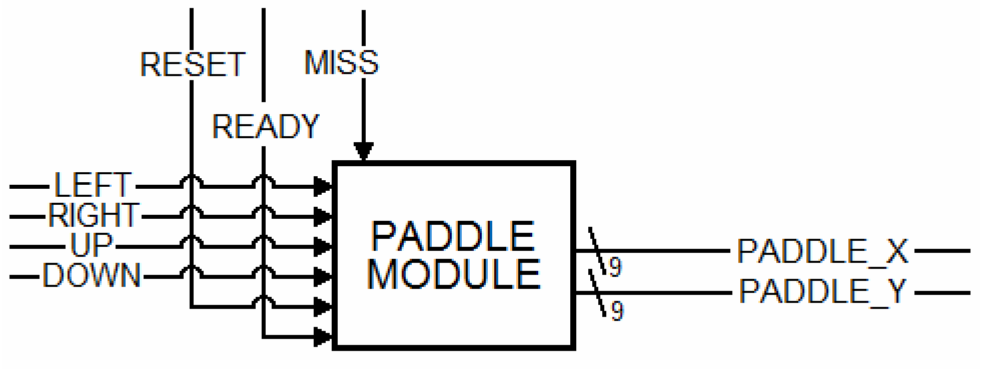


Figure 8: Paddle Module Block Diagram

The function of the Paddle Module is to move the paddle as directed by the player. The paddle will move in whatever direction the player directs until it reaches the edge of the space, at witch point it will stay there. The working functionality of this up, right, down, and left movement to the edge of the space is illustrated Images A though D in the test bench results on the next page (Figure 9). In addition to the ability to control the paddle in one direction at a time, some may find it useful to move diagonally (ie. In 2 directions at once). This ability is illustrated in Image E of Figure 9 (on the next page)— you can see $Paddle_x$ and $Paddle_y$ changing independently of each other and as instructed by the inputs. Also refer to Appendix C for the Verilog code and test bench for this module.

Just as with the puck, one of the indications that the player has lost is that the Paddle will stop moving whenever it misses the puck. However unlike the puck, the paddle takes the Miss input from the Game Module (discussed on **page FOO**) and disallows movement when the signal is high. Refer to Figure 9, Image E for the test bench visualization of this "Miss" functionality and to Appendix C for the desired output from the test bench for the Paddle.

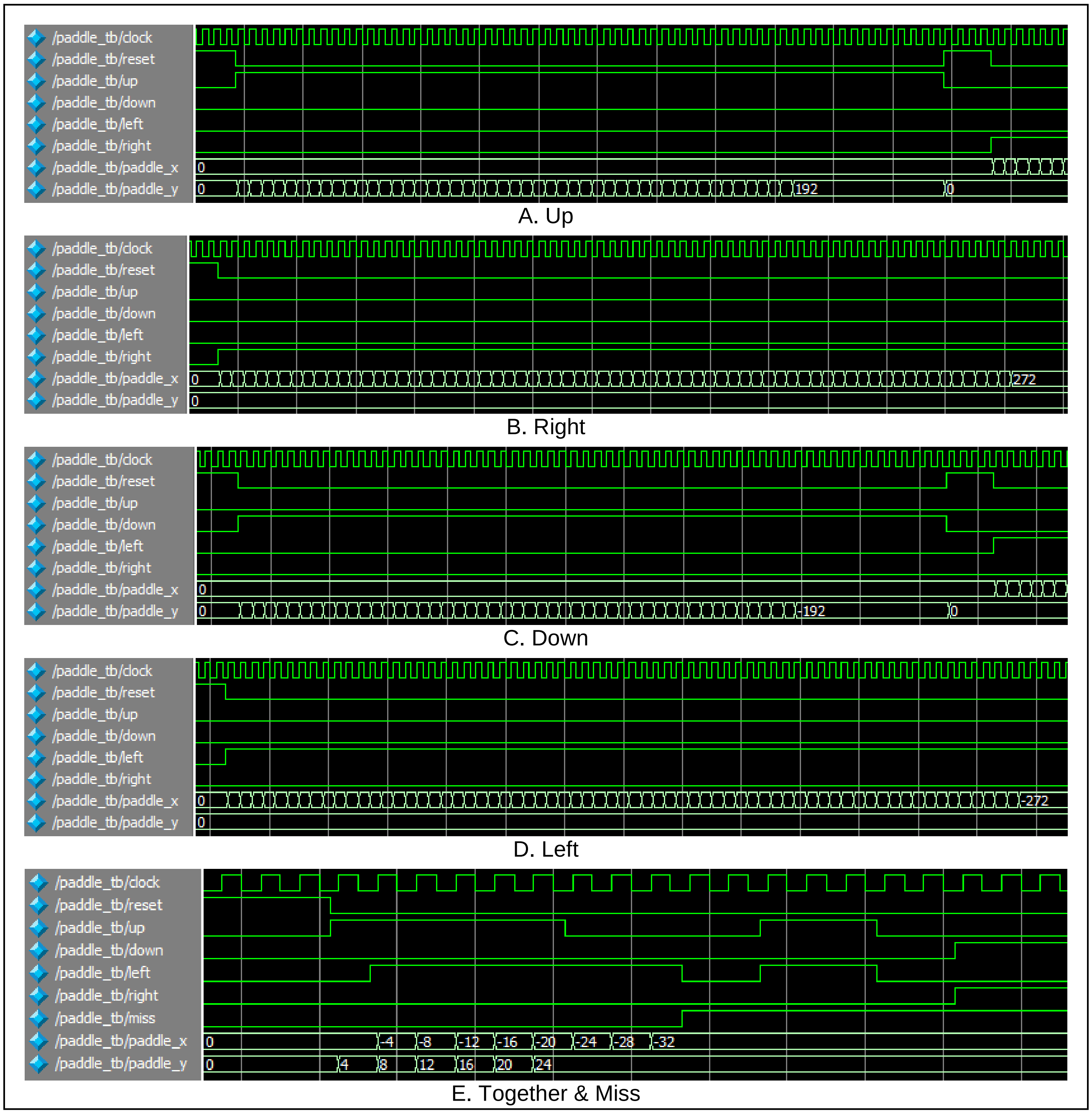


Figure 9: Paddle Test Bench Results

Game Module

The Game Module is where all of the work and logic for the game happens. As shown in Figure 10 on the next page, this module takes the inputs Reset, Ready, and the 9-bit x/y/z coordinates of the puck and paddle (Puck$_x$, Puck$_y$, Puck$_z$, Paddle$_x$, & Paddle$_y$). The Game Module will then use its internal logic to output the three 5-bit x, y, and z velocities (V$_x$, V$_y$, & V$_z$) and single bit Catch and Miss signals.

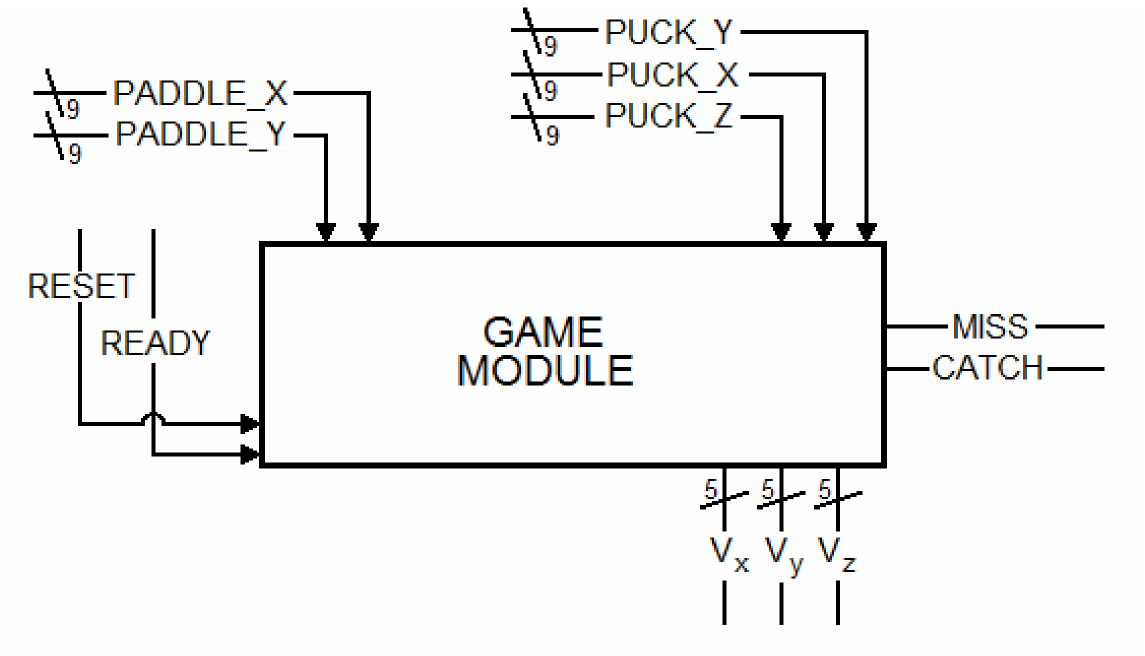


Figure 10: Game Module Block Diagram

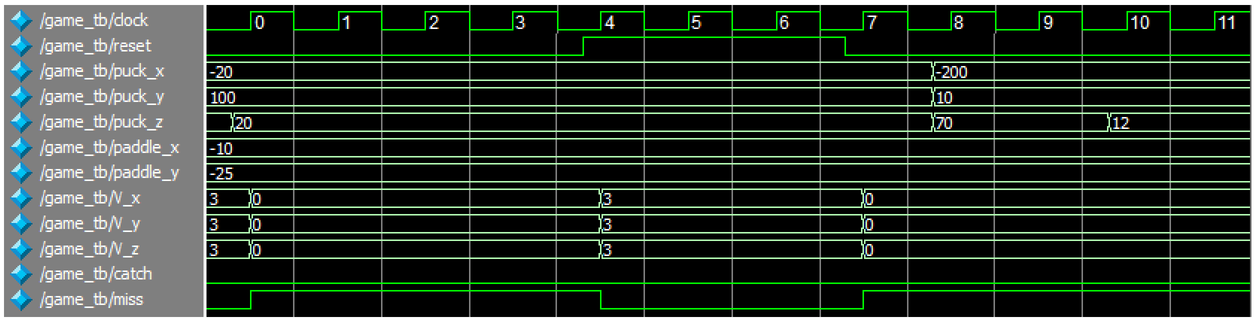
Within the function of a normal Pong game, all of the logic is used when the puck is at the same edge as the paddle — our implementation is no different. The first thing that the module looks for is that the puck is at the paddle (ie $Puck_z \leq 20$ = Radius of Puck) and if the puck is not at the paddle it will change nothing. If the puck is at the paddle, the internal logic will determine if it counts as a "catch" or a "miss" — enabling the appropriate signal and changing the velocities accordingly. Refer to Figure 11 (on the next page) for graphical representation of the Game Module Test Bench and Appendix D for the Verilog implementation of this module and test bench.

If the paddle misses the puck, the module will permanently enable "Miss" and set the x, y, & z velocities (V_x, V_y, & V_z) to zero until Reset is enabled. Referring to the test bench results in Image A of Figure 11 on the next page, you can verify that the module uses the following logic:

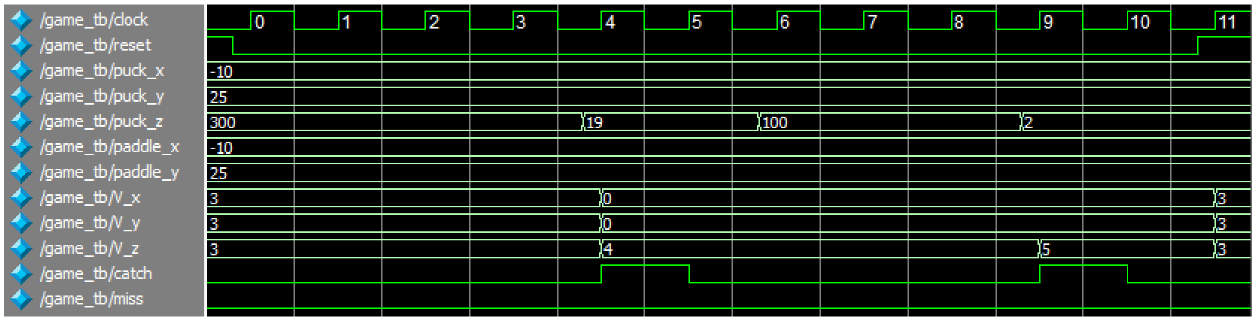
```
  (puck_x <= (paddle_x - 50))
| (puck_x >= (paddle_x + 50))
| (puck_y <= (paddle_y - 50))
| (puck_y >= (paddle_y + 50))
```

If the center of the puck does not fall within the area of the paddle, it counts as a miss and Miss becomes high. The right side of Image A also shows that if the game allowed the puck or paddle to move after Miss becomes enabled, nothing will change — Miss will stay high and the velocities will remain zero.**

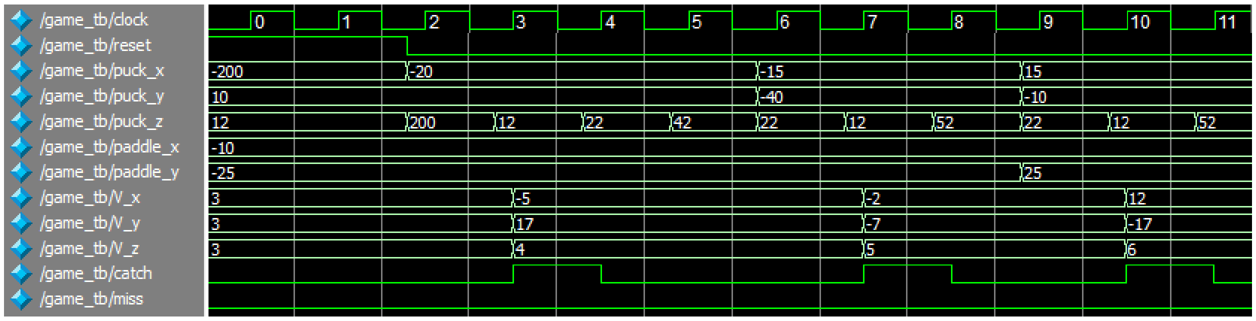
** Note that if all of the modules are functioning normally, this is a situation that cannot happen



A. Miss



B. Catch



C. Velocity Changing

Figure 11: Game Test Bench Results

If the paddle catches the puck, the module will enable a simple pulse on the Catch output. Looking at Image B of Figure 11 above, you can see that the output Catch is high for one ready cycle at each instant of the paddle catching the puck (clock cycles 4 and 9). The Game Module will also increase the z velocity (V_z) by one at each catch — again refer to Image B clock cycles 4 and 9 in Figure 11.

A catch also enables a change the x & y velocities (V_x & V_y), based on where on the Paddle the Puck hits. The Game Module looks at the difference between the center of the puck and center of the paddle and changes the velocity using the following formula:

```
((puck_y - paddle_y) / 2)
```

Image C of Figure 11 above, shows these changes in velocity as simulated in the test bench at clock cycles 3, 7, and 10. Also note that V_z changes as described above.

Score Module

The Score Module keeps track of the status of the game. As illustrated in Figure 12, below, the score module takes the inputs: Reset, Ready, Miss, and Catch. It then outputs an 8-bit Score and two 1-bit signals: Win and Lose. Each time the paddle “catches” the puck it increments and outputs the score, when the score reaches 63 catches the player has won the game and the Win signal is enabled. If at any point in time the player misses the puck, the module will enable the Lose signal. See Appendix E for the Verilog implementation of the Score Module.

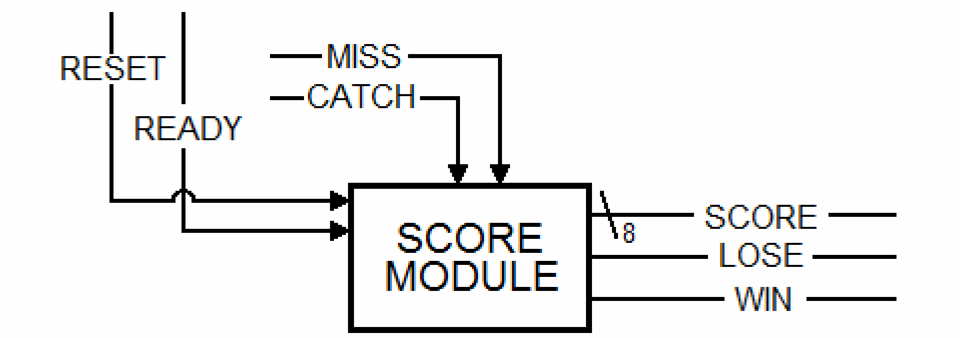
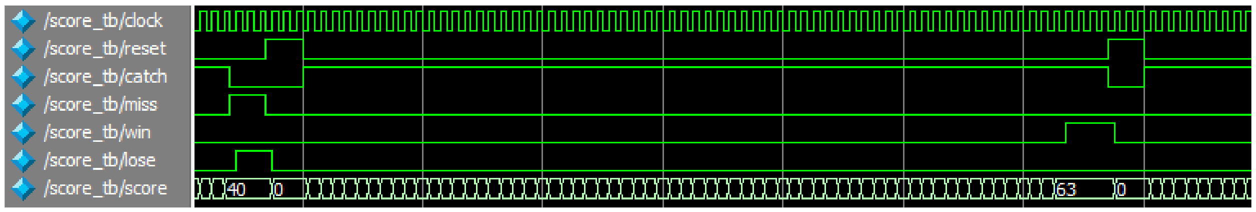
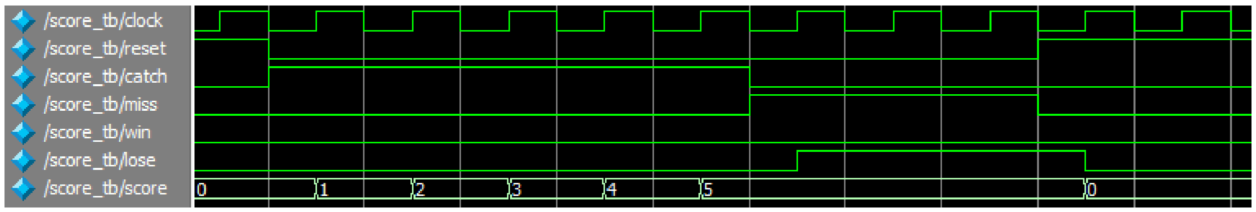


Figure 12: Score Module Block Diagram

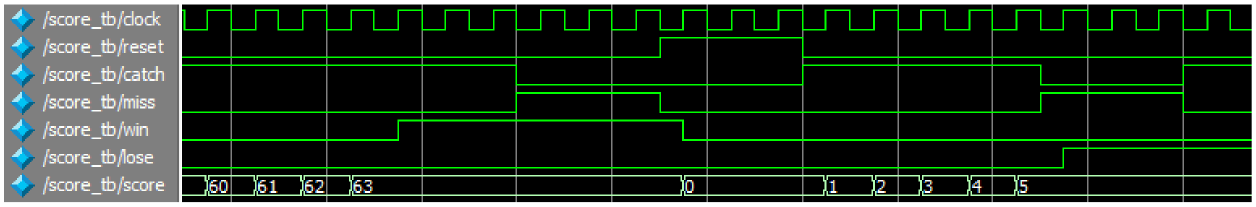
As expected from running the Score Module Test Bench (full code in Appendix E), Image B in Figure 13 (below) shows that the Score Module increments at every positive clock edge that Catch is enabled. Image B also shows that whenever there is a miss, the Lose signal is enabled and the score is frozen. After 63 catches without a miss, the player has ‘won’ and the Win signal is enabled — see Image A of Figure 13 below.



A. Win



B. Catch 5 & Miss



C. Stay Winning or Losing

Figure 13: Game Test Bench Results

Image C of Figure 13, on the previous page, illustrates that once win or lose have been asserted they cannot be changed until Reset is enabled. Although this is another instance of something that will not occur if the game is functioning normally, it is an important feature to keep things working — just in case something else isn't.

Description — Graphics

Richard Hughes

The graphics circuitry produced for this project uses a rendering pipe-line to produce 640x480 VGA graphics at 30 frames per second. As in Lab 5, the graphics pipeline takes the the hcount, vcount, hsync, vsync, and blank signals from the VGA module as input and provides a red-green-blue signal plus delayed phsync, pvsync, and pblank signals as output. However, where Lab 5 had a latency of two to four clock cycles, depending on the design, the pipelined ray-caster has a latency of over a hundred.

In order to allow the ray-casting modules to be pipelined, it was necessary for them to efficiently pass data from one to another. Each module takes in the direction of an incoming ray, and the distance and color of the last known intersection with the ray, and puts out the direction of the ray and the distance and color of the last known intersection with the ray. If the ray does not intersect the geometry, or intersects with the ray farther from the origin than a previous intersection, it outputs the same intersection and color that was input.

Rather than include the entire 24-bit color of the intersection, which would require 24 bits of register storage at each of the 100+ stages of the rendering pipeline and additional logic in each geometry-intersection module to calculate the color, I passed along two bits of data to select from a 'color palette' of three possible color functions. Using this color palette, the ray direction, and the ray distance, the 'Color Manager' module calculates the coordinates of the intersection, and then uses those coordinates as input for one of three color functions chosen by the color bits. This saves at least 3,168 bits of registers.

The diagram (Figure 14) on the next page summarizes the flow of data through the pipeline. The ray generator takes the hcount and vcount data and provides the rendering modules with a ray and initial intersection data that implies a non-intersection. The puck, wall, ceiling, floor, and paddle modules all process the ray and intersection data in turn, keeping them synchronized and overwriting the intersection data as appropriate as they go. The final step, the color manager, turns the ray data and intersection data in to a color to be written to a pixel on-screen. A pipeline delays the VGA synchronization signals to keep them in alignment with the rays and compensate for the latency of the rendering pipeline.

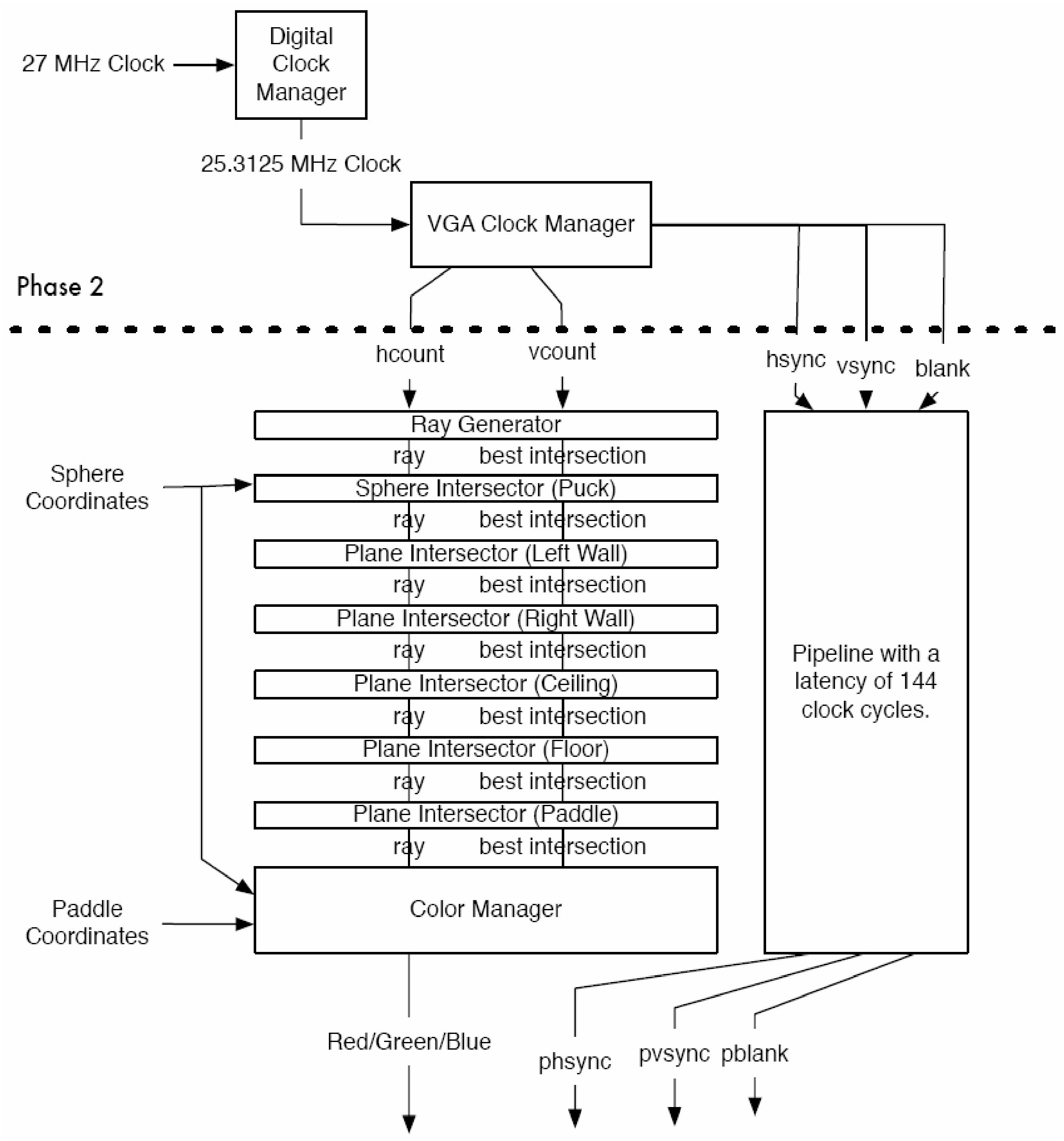


Figure 14: Overall Structure of the Pipelined Ray-Caster

RAY-TRACING AND RAY-CASTING

In order to understand how my code works, it's necessary to understand how ray-tracing and ray-casting work. Through geometry and algebra, it's possible to calculate whether a ray (defined as a point in space, plus a vector defining the direction the ray projects from that point) intersects a mathematically describable shape such as a sphere or plane, and where it intersects. In order to calculate the intersection of a ray and a shape, it must be possible to calculate if any given point is part of that shape. For example, the points of a sphere can be defined by the function "(X - Xc)^2 + (X - Xc)^2 + (X - Xc)^2 =

Radius^2" - i.e., that a given point is exactly calculable as either in the sphere or not. Further more, it must be possible to solve that function for t after you replace X, Y, and Z with "Xo + Xd * t", "Xo + Xd * t", and "Xo + Xd * t", respectively. The real-number solutions for t in that equation are the values of t for which that ray defined by the origin point Xo,Yo,Zo and the direction vector Xd,Yd,Zd intersects the object.

In ray tracing, you then create additional rays from the point of intersection. You send one to each light source, to check if it's shadowed or illuminating the point of intersection. If the object is translucent, one through the object at the angle defined by the refractive indices of the object and the air. If the object is reflective, you send one ray out at the reflected angle to see what color it strikes. The color contributes from these sources are then summed together according to their respective values (highly reflective objects add more from their reflection and less from their light source) and added to a baseline color representing the ambient light. Under this model, the number of rays needed to calculate the color for any given ray is unbounded, unless artificial cut-offs are imposed; a pair of mirrors facing each other can reflect indefinitely, creating an arbitrary amount of secondary rays.

Ray-casting can be interpreted as a special case of ray-tracing, where no objects are translucent or reflective, there are no light sources, and you use only ambient light. In other words, you never send out any rays from the initial point of intersection. As a result, the number of rays needed for any pixel is constant: one.

To keep the VGA signals coordinated with the pipeline, the hsync, vsync, and blank signals are passed through a relay of registers with the same throughput and (approximately) the same latency as the rendering pipeline, so that the coordinates of the pixels on the screen correspond to the hcount and vcount values passed to the rendering pipeline.

RAY-GENERATOR MODULE

To create the initial rays, and ensure each one corresponds to a pixel in a meaningful way, the rays all have the same origin and each one passes through a point on a flat rectangle in space analogous to the computer screen. Similar techniques were used in early artistic studies of perspective; see figure 0.

The data that a pipeline rendering module needs to take in - and thus the data that the ray generator needs to put out - are the Xd, Yd, and Zd directions of the ray, as well as the color and t-value (distance from origin) of the closest intersection so far. In order for us to generate this information, the ray generator module takes in the hcount and vcount values.

In some algorithms, the rays are 'normalized', their direction vectors set to a length of one with their proportions (and direction) preserved. Our algorithm does not do this, and instead uses ray vectors with integer components for Xd, Yd, and Zd. Because the origin point is at a fixed distance from the screen, and the screen is exactly perpendicular to the Z axis of our euclidean geometry, the Zd value is fixed. To take advantage of this, I hard-wired the Zd value in to all of my modules. The ray-generator module thus only needs to output the Xd and Yd values.

The ray generator module produces valid output as long as hcount and vcount are less than 640 and 480, respectively. Otherwise, it may produce garbage data. However, hcount and vcount are less than 640 and 480 for all pixels visible onscreen, so this is not an issue.

RAY FORMAT

Each ray is represented by an Xd and a Yd (both 10-bit 2's complement numbers). The previous intersection is represented by the t-value (a 14 bit integer, of which the most significant bit represents 2^1 and the least significant bit represents 2^{-12}) and the color value (a 2-bit integer where 0: no intersection, 1: Wall, Ceiling, or Floor, 2: Puck, and 3: the Paddle.) The t-value, which can be anywhere from 0 to 3 + 4095/4096, spans a range long enough to reach anywhere in the module

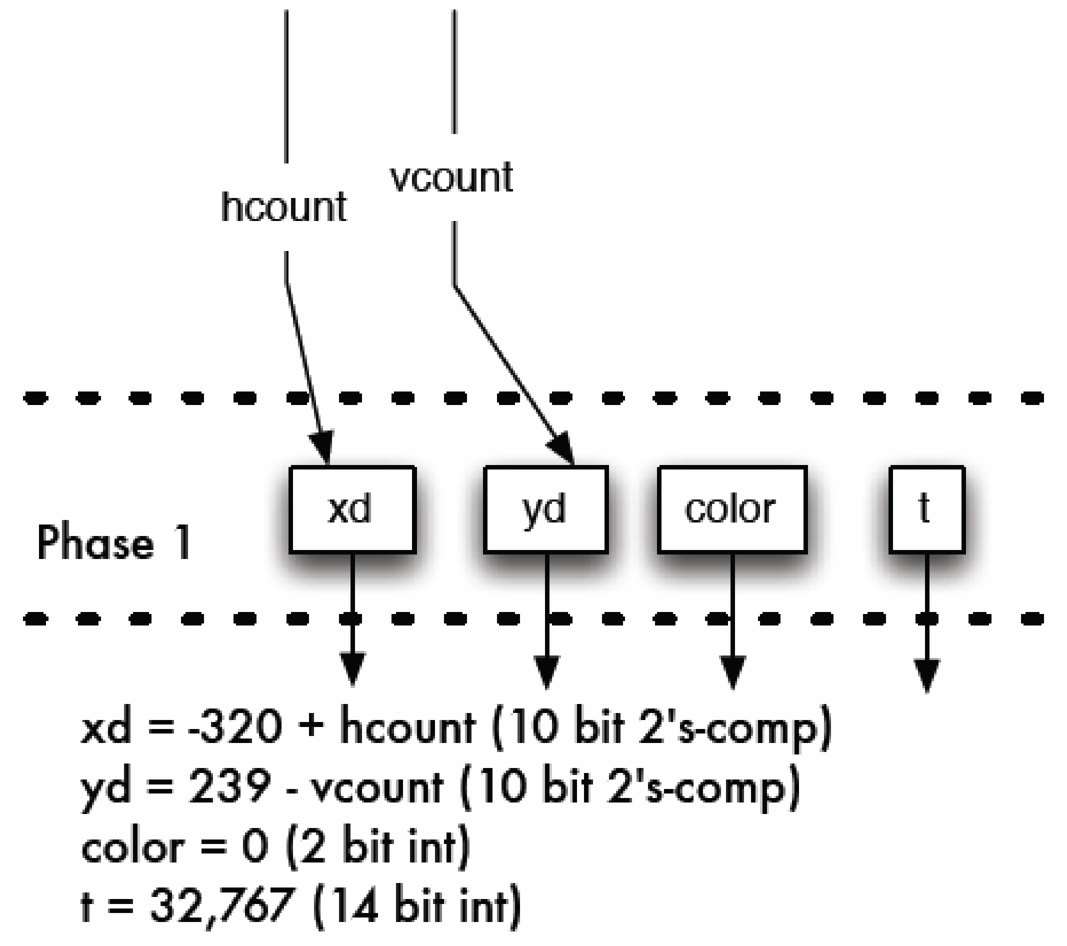


Figure 15: Ray Generator Figure

SPHERE-INTERSECTOR MODULE

The equation used to define a sphere is:

$$(X - X_c)^2 + (Y - Y_c)^2 + (Z - Z_c)^2 = Radius^2$$

Where X, Y, and Z are any given point, Xc, Yc, and Zc are the center of the sphere, and Radius is the radius of the sphere. When we substitute $X_0 + X_d * t$, etc, for X, Y, and Z, this becomes:

$$(X_0 + X_d * t - X_c)^2 + (Y_0 + Y_d * t - Y_c)^2 + (Z_0 + Z_d * t - Z_c)^2 = Radius^2$$

We can isolate t^2, t, and 1 in this equation with 0 on the right side to get an equation we can solve by the quadratic equation:

$$A*t^2 + B*t + C = 0$$
$$A = X_d^2 + Y_d^2 + Z_d^2$$
$$B = 2 * (X_d * (X_0 - X_c) + Y_d * (Y_0 - Y_c) + Z_d * (Z_0 - Z_c))$$
$$C = (X_0 - X_c)^2 + (Y_0 - Y_c)^2 + (Z_0 - Z_c)^2 - S_r^2$$
$$t = (-B +/- (B^2 - 4*A*C)^{1/2}) / 2A$$

The module has the radius of the sphere hard-wired in, but it must receive the sphere's center as input. The sphere's center is defined by an X, Y, and Z value which are 10, 10, and 11 bit 2's complement signed integers, respectively.

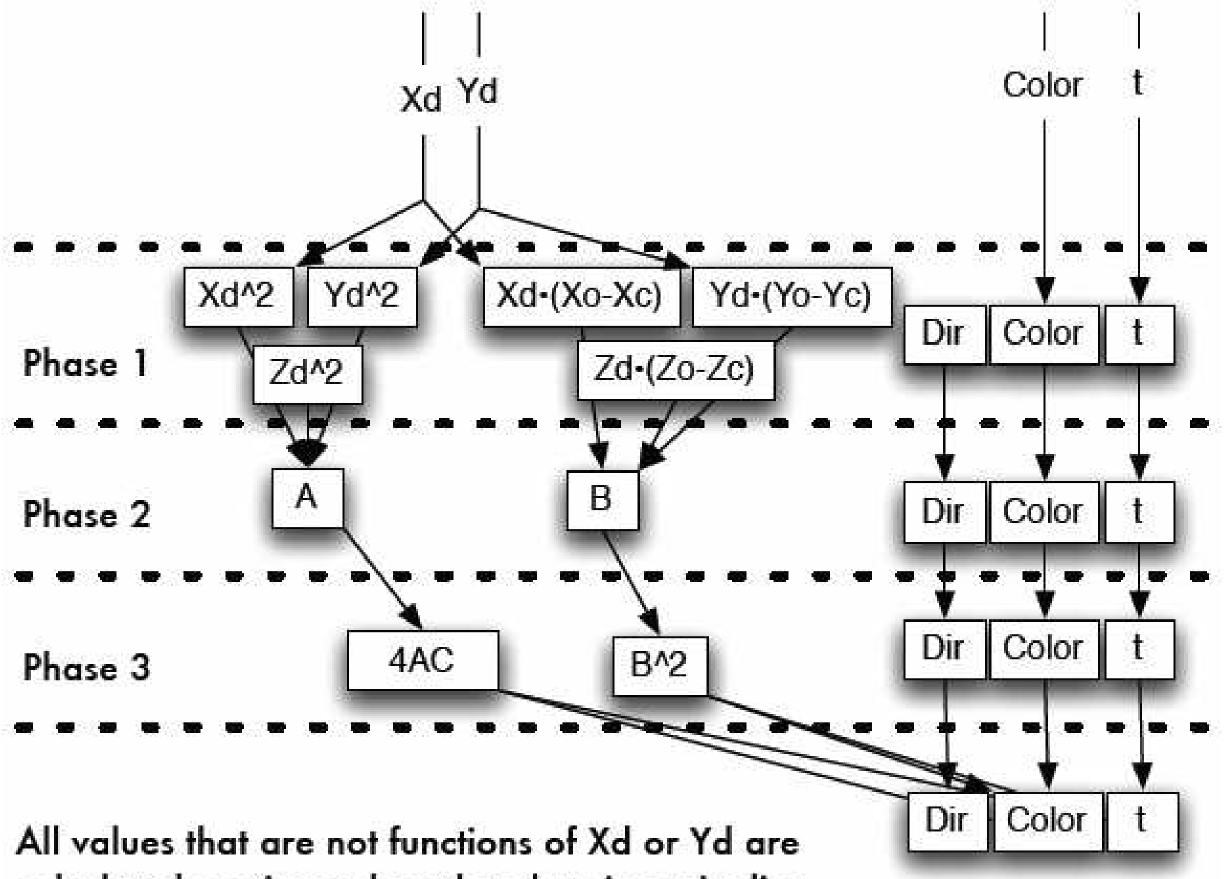
Normally, we would need to use a square root module in order to calculate $(B^2 - 4*A*C)^{1/2}$. However, we cheat; we know that the sphere is always in front, so we don't bother to calculate anything except whether or not $(B^2 - 4*A*C)$ is negative. If it is negative, there is no real solution and so there is no intersection. In this instance, the module outputs whatever color and t value it was originally provided. If it is not negative, there is a solution, and thus there is an intersection, and it's always the first intersection. In this instance, the module outputs color = 2 and t = 4095. The lowest possible t value that can be created by the other geometry in the scene is 4096, so the sphere is always in front. The module always outputs the same X_d and Y_d it received at the same time it outputs the corresponding color and t values.

***BUGS** — ie. IT DOESN'T WORK*

Unfortunately, the sphere intersector module does not function properly. While the output for a successful or unsuccessful intersection is correct, intersections do not occur as they should. The cross section on the screen is not circular, but warps in appearance like a hyperbolic shape. Whatever quadratic equation the module solves, it isn't a sphere.

Investigation and testing showed that the module works effectively in simulation, but fails in implementation. The most obvious possibility is that there is a timing problem, as the simulation does not reveal those. Some possible avenues of solution are replacing the behavior-description verilog multiplier with a pipelined multiplier module, but this may or may not function. Alternatively, the module is small enough that it would be practical to scrap it and start over completely, ideally to avoid whatever minor error led to the flawed behavior in the second attempt. Ultimately, the failure of this module is a mystery to me.

The diagram below illustrates the pipelining for the sphere intersector.



All values that are not functions of Xd or Yd are calculated continuously rather than in a pipeline.

If B^2 > 4AC, then the discriminant in the quadratic equation is positive, meaning there are real solutions and real intersections.

If there is an intersection, we assume it's the best intersection and output color = 2 and t = 4095. (4095 / 4096 is less than one, so this is 'before' the screen, and thus closer than any other object.)

Figure 16: Sphere Intersector Diagram

PLANE-INTERSECTOR MODULE

The equation used to define a plane is:

$$A x + B y + C z + D = 0$$

Where (A,B,C) is a vector defining the normal of the plane, and D is how close the plane comes to the origin (positive values indicating the normal of the plane points towards the origin, negative values indicating the plane points away.) If we substitute the ray equations for X, Y, and Z and solve for t, we get:

$$A(X_0 + X_d * t) + B(Y_0 + Y_d * t) + (Z_0 + Z_d * t) + D = 0$$
$$t = - (A * X_0 + B * Y_0 + C * Z_0 + D) / (A * X_d + B * Y_d + C * Z_d)$$

Unlike in the sphere, the plane is in a constant position, so A, B, C, and D are all hardwired. This makes it easy to calculate the value of $(A * X_d + B * Y_d + C * Z_d)$. $(A * X_0 + B * Y_0 + C * Z_0 + D)$ is constant, because the plane and the origin do not move. - indeed, since the origin has $X_0 = 0$ and $Y_0 = 0$, we can simplify it to $(C * Z_0 + D)$ and save time. However, dividing $(A * X_0 + B * Y_0 + C * Z_0 + D)$ by $(A * X_d + B * Y_d + C * Z_d)$ takes 25 clock cycles with a pipelined divider. Calculating the t value thus takes 27 clock cycles.

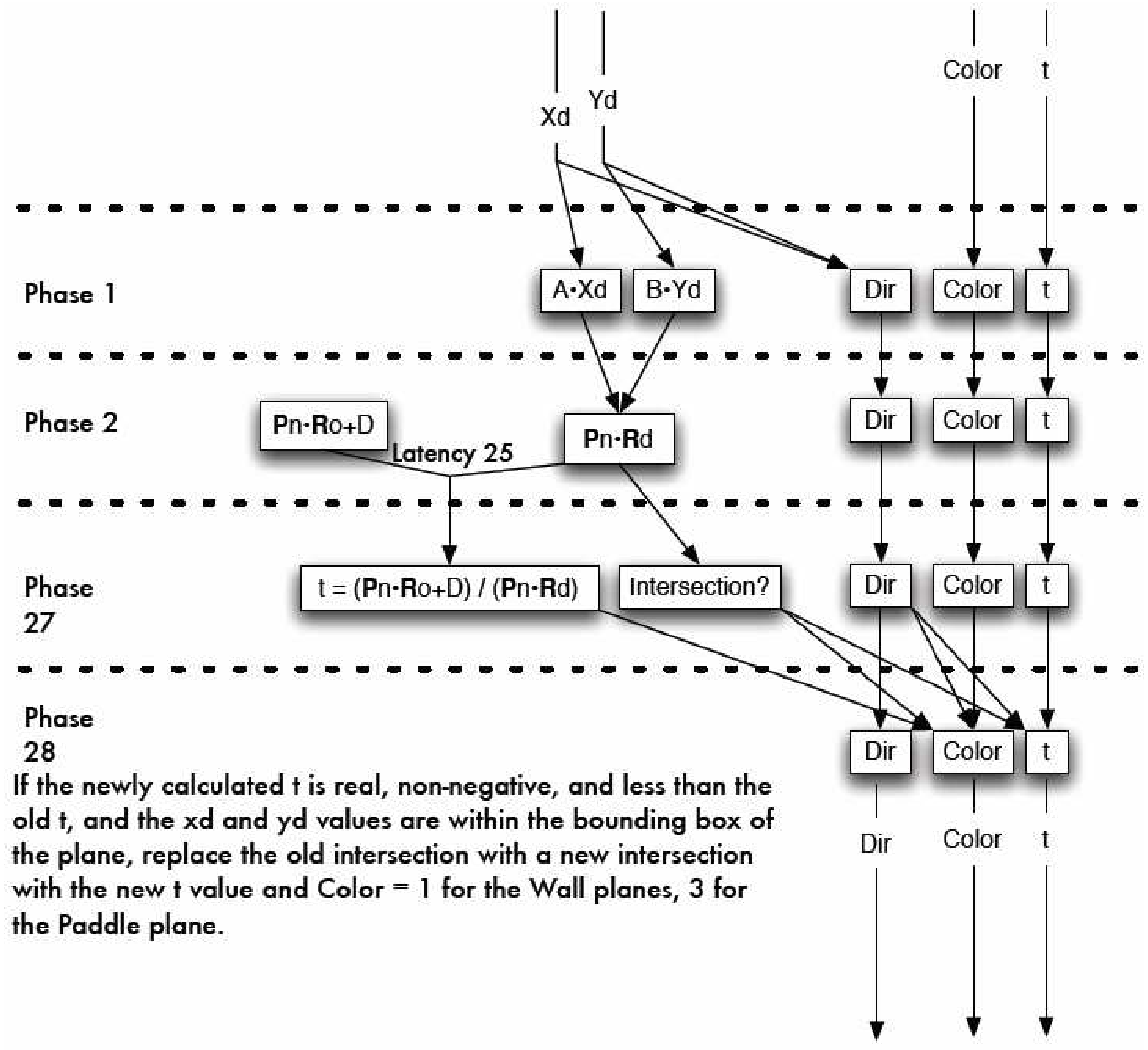
***BUGS** — GOING RIGHT ROUND*

When I first tested the plane intersector module, the back paddle didn't seem to be working properly. I could see the other four planes extending off in to infinity as parallel lines, and I couldn't fathom why. Later, I realized that the reason for the problem was due to the limited number of bits in the t value - any t equal to or greater than 2^14 came out modulo 2^14, because t only had 14 bits. As a result, extremely distant objects such as the planes extending in to the distance came out with lower t values than much closer objects, and so they were errantly drawn in front.

Because the four planes extended infinitely to the horizon under my geometry model, no number of bits in t would fix this problem. Additionally, every bit of storage in t would need to be duplicated at every point in the pipeline, which could become expensive quickly. I solved the problem with the cheap hack of defining a 'rendering box' for each plane, maximum and minimum xd and yd values that could intercept them. All rays that fell outside that boundary automatically missed. I set the bounding boxes to block off the areas of the screen where the t values began to wrap around, and the problem no longer appeared.

Another bug I experienced while developing and testing the plane intersector module was not in the code, but in the FPGA. When the FPGA was reprogrammed without being power cycled first (i.e., turn it off and turn it on again), graphical glitches would appear in the screen. Programming in to a 'fresh' FPGA eliminates these flaws.

The diagram on the next page illustrates the pipelining for the plane intersector.



Xd
Yd
Color
t
Phase 1
A•Xd
B•Yd
Dir
Color
t
Phase 2
Pn•Ro+D
Latency 25
Pn•Rd
Dir
Color
t
Phase 27
t = (Pn•Ro+D) / (Pn•Rd)
Intersection?
Dir
Color
t
Phase 28
If the newly calculated t is real, non-negative, and less than the old t, and the xd and yd values are within the bounding box of the plane, replace the old intersection with a new intersection with the new t value and Color = 1 for the Wall planes, 3 for the Paddle plane.
Dir
Color
t
Dir
Color
t

Figure 17: Plane Intersector Diagram

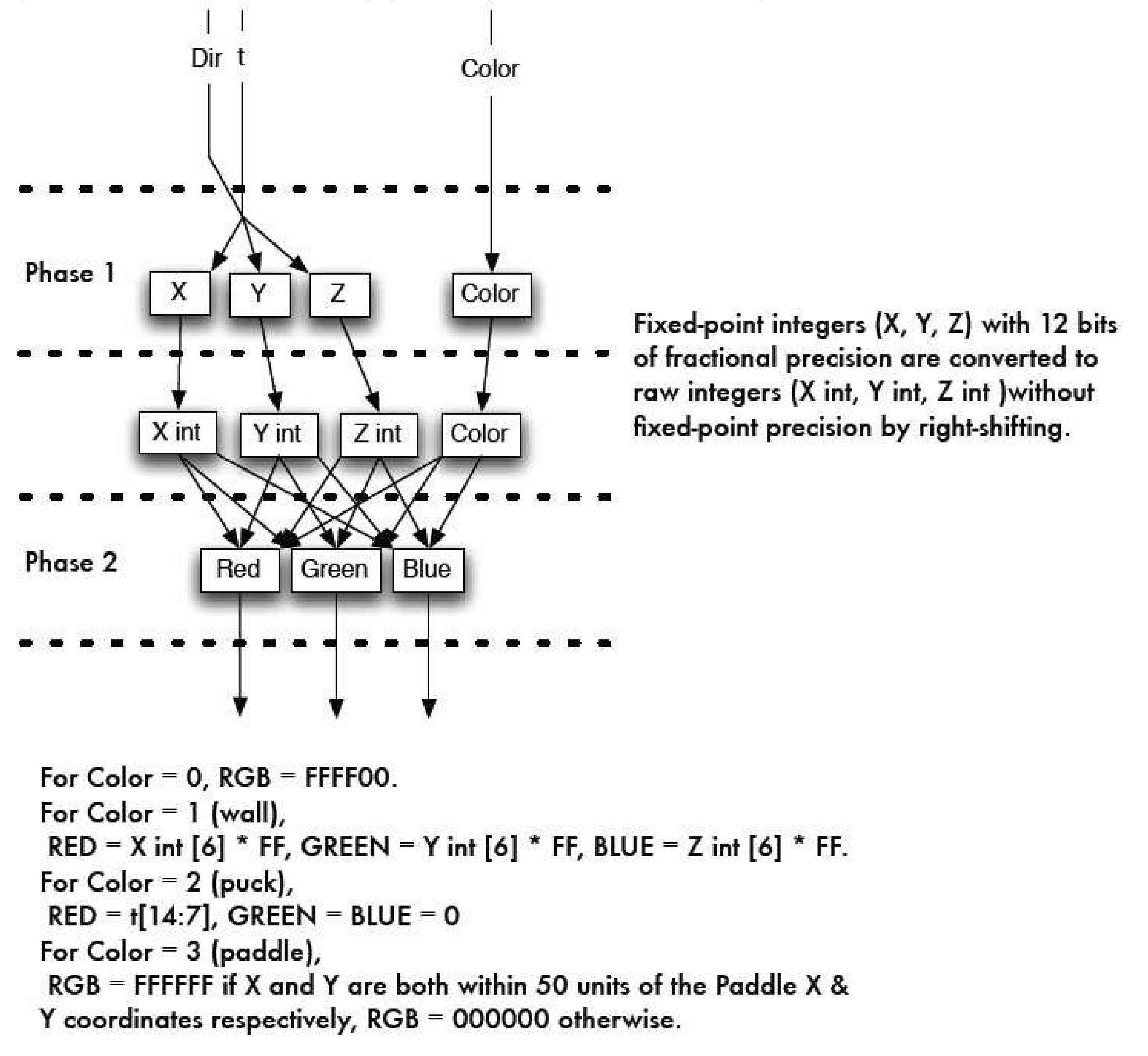
COLOR MANAGER

The color manager takes the intersection information (direction, t, and palette code) and the paddle and puck coordinates. In the first phase, it uses the direction and t to calculate the position of the intersection in XYZ coordinates. In the second phase, it removes the 12 least significant bits of the results, which represent fractional value. In the third phase, it takes the palette code and chooses one of four color functions:

- A checkerboard for the walls,
- A white square for the paddle,
- A flat pink for the puck,
- A bright yellow for a 'miss'.

The output is an RGB value appropriate for the pixel passed in.

The diagram blow illustrates the pipelining of the color manager module.


Dir
t
Color
Phase 1
X
Y
Z
Color
X int
Y int
Z int
Color
Phase 2
Red
Green
Blue
Fixed-point integers (X, Y, Z) with 12 bits of fractional precision are converted to raw integers (X int, Y int, Z int)without fixed-point precision by right-shifting.

For Color = 0, RGB = FFFF00.
For Color = 1 (wall),
RED = X int [6] * FF, GREEN = Y int [6] * FF, BLUE = Z int [6] * FF.
For Color = 2 (puck),
RED = t[14:7], GREEN = BLUE = 0
For Color = 3 (paddle),
RGB = FFFFFF if X and Y are both within 50 units of the Paddle X & Y coordinates respectively, RGB = 000000 otherwise.

Figure 18: Color Managing Diagram

Conclusion

The design presented here is a product of many compromises between our limitations and our accomplishments. Among the features completed here are functioning physics (or backend) modules, a full ModelSim testing suite for the backend, real time pipelined ray-casting with texture-palette color management, and a functioning ray-plane intersection module. Unfortunately, the sphere rendering module is not functional. Further iterations to the design could repair and improve the sphere intersector module to allow for a functional sphere intersection, and later, a sphere intersector that provides not only a boolean "intersected / did not intersect" test but provides the exact point of intersection. By repairing the sphere intersector and synchronizing the coordinate input more carefully, it should be possible to quickly improve the game to a playable level.

Appendix A: Ready Module & Test Bench

```
//   READY MODULE
// Turn the 65MHz Clock into a 30Hz Signal

module ready_module
             (input clock, reset,
              output reg ready,
              output reg [18:0] counter);

always @(posedge clock)
   if (reset)
    begin
      counter <= 0;
      ready <= 0;
    end
   else
    begin
      if (counter == 2166667)
//    if (counter == 21)          // for testing purposes
        begin
          counter <= 0;
          ready <= 1;
        end
      else if (counter == 0)
        begin
          ready <= 0;
          counter <= counter + 1;
        end
      else
        counter <= counter + 1;
    end   // else

endmodule


//   READY MODULE TEST BENCH
// for testing the ready module

`timescale 1 ns / 1 ps

module ready_tb ();

reg clock, reset;
wire ready;
wire [18:0] counter;

initial begin
  clock = 0;
  forever #5 clock = ~clock;  // goes high every #10
end

initial begin
  reset = 1;
  #33;
  reset = 0;
  #5000
  reset = 1;
  #30;
  reset = 0;
  #50
  $stop();
  end

ready_module readysignal(.clock(clock),.reset(reset),
              .ready(ready),.counter(counter));
endmodule
```

Appendix B: Puck Module & Test Bench

```
//    PUCK MODULE
// the puck moves around the screen based on x, y, & z input velocities.  It
// "bounces" off of "walls" when it reaches the edge of our space.

module puck_module (
   input clock,                       // 24MHz clock
   input reset,                       // 1 to initialize module
   input signed [5:0]  V_x,           // puck horizontal speed in pixels/tick
   input signed [5:0]  V_y,           // puck vertical speed in pixels/tick
   input        [5:0]  V_z,           // puck depth speed in pixels/tick

   output reg signed [9:0] puck_x,    // puck's horizontal position
   output reg signed [9:0] puck_y,    // puck's vertical position
   output reg signed [9:0] puck_z     // puck's depth position
   );

reg xMovement;
reg yMovement;
reg zMovement;

always @(posedge clock)
 begin

if (reset)
 begin
      puck_x <= 0;
      puck_y <= 0;
      puck_z <= 320;
      xMovement <= 1;                       // moving right
      yMovement <= 1;                       // moving down
      zMovement <= 1;                       // moving forward
 end //reset

else
 begin //Go Go Puck

      if (xMovement)                        // horizontal movement
       begin
            if (puck_x >= 300)
                  xMovement <= 0;
            else
                  puck_x <= puck_x + V_x;
       end  // xMovement
      else  // !xMovement
       begin
            if (puck_x <= - 300)
                  xMovement <= 1;
            else
                  puck_x <= puck_x - V_x;
       end // !xMovement

      if (yMovement)                        // vertical movement
       begin
            if (puck_y >= 220)
                  yMovement <= 0;
            else
                  puck_y <= puck_y + V_y;
       end  // yMovement
      else  // !yMovement
       begin
            if (puck_y <= - 220)
                  yMovement <= 1;
            else
                  puck_y <= puck_y - V_y;
       end // !yMovement
      if (zMovement)                        // depth movement
       begin
            if (puck_z >= 620)
                  zMovement <= 0;
            else
                  puck_z <= puck_z + V_z;
       end  // zMovement
      else  // !zMovement
       begin
            if (puck_z <= 20)
                  zMovement <= 1;
            else
                  puck_z <= puck_z - V_z;
       end // !zMovement

 end //Go Go Puck

end //aways block
endmodule
```

// PUCK TEST MODULE   ** test module is for a larger scene

```
// for testing the puck module

`timescale 1 ns / 1 ps

module puck_tb ();

reg clock, reset;
reg signed [5:0] V_x;
reg signed [5:0] V_y;
reg        [5:0] V_z;
wire signed [10:0] puck_x;
wire signed [10:0] puck_y;
wire signed [10:0] puck_z;

initial begin
  clock = 0;
  forever #5 clock = ~clock;  // goes high every #10
end

initial begin
  reset = 1;
  #33;             // puck_x=0, puck_y=0, puck_z=500
  reset = 0;
  V_x = 0;
  V_y = 0;
  V_z = 0;
  #300             // puck_x=0, puck_y=0, puck_z=500
//-------------------------------------------------------------reset-------------
  reset = 1;
  #30;             // puck_x=0, puck_y=0, puck_z=500
  reset = 0;
  V_x = 5;
  V_y = 0;
  V_z = 0;
  #5000            // puck_x="count" by 2 +/- 472, puck_y=0, puck_z=500
//-------------------------------------------------------------reset-------------
  reset = 1;
  #30;             // puck_x=0, puck_y=0, puck_z=500
  reset = 0;
  V_x = 0;
  V_y = 5;
  V_z = 0;
  #3000            // puck_x=0, puck_y="count" by 2 +/- 344, puck_z=500
//-------------------------------------------------------------reset-------------
  reset = 1;
  #30;             // puck_x=0, puck_y=0, puck_z=500
  reset = 0;
  V_x = 0;
  V_y = 0;
  V_z = 5;
  #5000            // puck_x=0, puck_y=0, puck_z="count" by 20-1004
//-------------------------------------------------------------reset-------------
  reset = 1;
  #30;             // puck_x=0, puck_y=0, puck_z=500
  reset = 0;
  V_x = 5;
  V_y = 5;
  V_z = 5;
  #5000            // all "count" by 2... x=+/- 472, y=+/- 344, z=20-1004

  $stop();
end

puck_module puck(.clock(clock),.reset(reset),
                 .V_x(V_x),.V_y(V_y),.V_z(V_z),
                 .puck_x(puck_x),.puck_y(puck_y),.puck_z(puck_z));

endmodule
```

Appendix C: Paddle Module & Test Bench

```
//    PADDLE MODULE
// The paddle moves around back wall based on x & y inputs.  The inputs are up,
// down, left & right and the paddle moves accordingly.


module paddle_module (
   input clock,                        // 24MHz clock
   input reset,                        // 1 to initialize module
   input up,                           // 1 when paddle should move up
   input down,                         // 1 when paddle should move down
   input left,                         // 1 when paddle should move left
   input right,                        // 1 when paddle should move right
   input miss,                         // paddle doesn't catch the puck

   output reg signed [9:0] paddle_x,   // paddle's horizontal position
   output reg signed [9:0] paddle_y ); // paddle's vertical position

always @(posedge clock)
 begin

if (reset)                             // reset
 begin
      paddle_x <= 0;
      paddle_y <= 0;
 end
else if (miss)                         // miss
 begin
      paddle_x <= paddle_x;
      paddle_y <= paddle_y;
 end
else
 begin //gogo paddle
                                       // Horizontal Movement
      if (left)
       begin
            if (paddle_x <= - 270)
                  paddle_x <= paddle_x;
            else
                  paddle_x <= paddle_x - 4;
       end // left
      else if (right)
       begin
            if (paddle_x >= 270)
                  paddle_x <= paddle_x;
            else
                  paddle_x <= paddle_x + 4;
       end //down
                                       // Vertical Movement
      else if (up)
       begin
            if (paddle_y >= 190)
                  paddle_y <= paddle_y;
            else
                  paddle_y <= paddle_y + 4;
       end // up
      else if (down)
       begin
            if (paddle_y <= - 190)
                  paddle_y <= paddle_y;
            else
                  paddle_y <= paddle_y - 4;
       end //down
      else
       begin
            paddle_x <= paddle_x;
            paddle_y <= paddle_y;
       end
 end //gogo paddle

end //always statement
endmodule
```

```
//  PADDLE TEST MODULE                             ** test module is for a larger scene
// for testing the paddle module

`timescale 1 ns / 1 ps

module paddle_tb ();

reg clock, reset;
reg up, down, left, right;
wire signed [10:0] paddle_x;
wire signed [10:0] paddle_y;

initial begin
  clock = 0;
  forever #5 clock = ~clock;   // goes high every #10
end

initial begin
  reset = 1;
  up = 0;
  down = 0;
  left = 0;
  right = 0;
  #43;
  reset = 0;
  up = 1;
  down = 0;
  left = 0;
  right = 0;
  #1000                        // paddle_x = 0, paddle_y => 284
  right = 1;
  up = 0;
  #1500                        // paddle_x => 412, paddle_y = 284
  down = 1;
  right = 0;
  #2000                        // paddle_x = 0, paddle_y => -284
  left = 1;
  down = 0;
  #3000                        // paddle_x => -412, paddle_y = 0
  reset = 1;
  #30;                         // paddle_x = 0, paddle_y =0

  $stop();
end

paddle_module game(.clock(clock),.reset(reset),
                    .up(up),.down(down),.left(left),.right(right),
                    .paddle_x(paddle_x),.paddle_y(paddle_y));

endmodule
```

Appendix D: Game Module & Test Bench

```
//    GAME MODULE
// Impliments the interactions between the puck and the paddle

module game_module (
   input clock,                        // 24MHz clock
   input reset,                        // 1 to initialize module
   input signed [9:0] puck_x,          // puck's horizontal position
   input signed [9:0] puck_y,          // puck's vertical position
   input signed [9:0] puck_z,          // puck's depth position
   input signed [9:0] paddle_x,        // paddle's horizontal position
   input signed [9:0] paddle_y,        // paddle's verdical position

   output reg signed [5:0] V_x,        // puck's absolute horiz. velocity
   output reg signed [5:0] V_y,        // puck's absolute vertical velocity
   output reg        [5:0] V_z,        // puck's absolute depth velocity
   output reg catch,                   // paddle catches puck
   output reg miss,                    // paddle doesn't catches puck
   output reg signed [9:0] V_xDelta,   // x difference between puck & paddle
   output reg signed [9:0] V_yDelta ); // y difference between puck & paddle

always @(posedge clock)
 begin

if (reset)                              // reset game
 begin
      miss <= 0;
      catch <= 0;
      V_x <= 3;
      V_y <= 3;
      V_z <= 3;
      V_xDelta <= 0;
      V_yDelta <= 0;
 end // reset
else if (miss)                          // if missed puck...stay missed
 begin
      miss <= 1;
      V_x <= 0;
      V_y <= 0;
      V_z <= 0;
 end //miss
else if (catch)
      catch <= 0;

else if (puck_z <= 20)                  // puck is at the back of the screen
 begin

      if ((puck_x <= (paddle_x - 50))
        | (puck_x >= (paddle_x + 50))
        | (puck_y <= (paddle_y - 50))
        | (puck_y >= (paddle_y + 50)))  // puck does not hit paddle
       begin //miss puck
            miss <= 1;
            V_x <= 0;
            V_y <= 0;
            V_z <= 0;
       end //miss puck
      else                              // puck hits the paddle
       begin //catch puck
            catch <= 1;
            V_xDelta <= (puck_x - paddle_x);
            V_yDelta <= (puck_y - paddle_y);
            V_x <= ((puck_x - paddle_x) / 2);
            V_y <= ((puck_y - paddle_y) / 2);
            V_z <= V_z + 1;
       end //catch puck
 end //puck_z <=40
```

```
else                   // not reset, has not missed, or puck is not @ the back
 begin
      miss <= miss;
      catch <= catch;
      V_x <= V_x;
      V_y <= V_y;
      V_z <= V_z;
 end //else

 end // always block
endmodule
```

```
//  GAME TEST MODULE                          ** test module is for a larger scene
// for testing the game module

`timescale 1 ns / 1 ps

module game_tb ();

reg clock, reset;
reg signed [10:0] puck_x;
reg signed [10:0] puck_y;
reg signed [10:0] puck_z;
reg signed [10:0] paddle_x;
reg signed [10:0] paddle_y;
wire [5:0] V_x;
wire [5:0] V_y;
wire [5:0] V_z;
wire [10:0] V_xDelta;
wire [10:0] V_yDelta;
wire catch, miss;

initial begin
  clock = 0;
  forever #5 clock = ~clock;  // goes high every #10
end

initial begin
  reset = 1;
  #43                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  reset = 0;
  #20                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_x = -10;
  puck_y = 25;
  puck_z = 560;
  #10                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_x = -10;
  puck_y = 25;
  puck_z = 560;
  paddle_x = -10;
  paddle_y = 25;
  #10                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_z = 19;
  #10                    // V_x= 0, V_y= 0, V_z= 5, catch= 1, miss= 0
  #10                    // V_x= 0, V_y= 0, V_z= 5, catch= 0, miss= 0
  puck_z = 777;
  #30                    // V_x= 0, V_y= 0, V_z= 5, catch= 0, miss= 0
  puck_z = 2;
  #10                    // V_x= 0, V_y= 0, V_z= 6, catch= 1, miss= 0
  #10                    // V_x= 0, V_y= 0, V_z= 6, catch= 0, miss= 0
// ---------------------------------------------------------------reset----------
  reset = 1;
  #30                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  reset = 0;
  puck_x = -20;
  puck_y = 100;
  puck_z = 560;
  paddle_x = -10;
  paddle_y = -25;
  #20                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_z = 20;
  #20                    // V_x= 0, V_y= 0, V_z= 0, catch= 0, miss= 1
// -----------------------------------------------------------reset----------
  reset = 1;
  #30                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  reset = 0;
  #10                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_x = -200;
  puck_y = 10;
  puck_z = 560;
  paddle_x = -10;
  paddle_y = -25;
  #20                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_z = 12;
  #20                    // V_x= 0, V_y= 0, V_z= 0, catch= 0, miss= 1
// -----------------------------------------------------------reset----------
  reset = 1;
  #30                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  reset = 0;
  puck_x = -20;
  puck_y = 10;
  puck_z = 560;
  paddle_x = -10;
  paddle_y = -25;
  #10                    // V_x= 4, V_y= 4, V_z= 4, catch= 0, miss= 0
  puck_z = 12;
  #10                    // V_x= -10, V_y= 15, V_z= 5, catch= 1, miss=0
  puck_z = 22;
  #10                    // V_x= -10, V_y= 15, V_z= 5, catch= 0, miss=0
  puck_z = 42;
  #10                    // V_x= -10, V_y= 15, V_z= 5, catch= 0, miss=0
  puck_x = -15;
  puck_y = -40;
  puck_z = 22;
  paddle_x = -10;
  paddle_y = -25;
  #10                    // V_x= -10, V_y= 15, V_z= 5, catch= 0, miss=0
  puck_z = 12;
  #10                    // V_x= -5, V_y= -15, V_z= 6, catch= 1, miss=0
  puck_z = 52;
  #10                    // V_x= -5, V_y= -15, V_z= 6, catch= 0, miss=0
  puck_x = 15;
  puck_y = -10;
  puck_z = 22;
  paddle_x = -10;
  paddle_y = 25;
  #10                    // V_x= -5, V_y= -15, V_z= 6, catch= 0, miss=0
  puck_z = 12;
  #10                    // V_x= 25, V_y= -35, V_z= 7, catch= 1, miss=0
  puck_z = 52;
  #10                    // V_x= 25, V_y= -35, V_z= 7, catch= 0, miss=0

  $stop();
end

game_module gamer(.clock(clock),.reset(reset),
                  .puck_x(puck_x),.puck_y(puck_y),.puck_z(puck_z),
                  .paddle_x(paddle_x),.paddle_y(paddle_y),
                  .V_x(V_x),.V_y(V_y),.V_z(V_z),
                  .catch(catch),.miss(miss),
                  .V_xDelta(V_xDelta),.V_yDelta(V_yDelta));

endmodule
```

Appendix E: Score Module & Test Bench

```
//    SCORE MODULE
// counts the number of times that the paddle "catches" the puck.  It
// also creates the win and lose signals for the game.

module score_counter(
      input clock,
      input reset,
      input catch,
      input miss,

      output reg win,
      output reg lose,
      output reg [5:0] score);

always @(posedge clock)
 if (reset)                     // reset
  begin
      win <= 0;
      lose <=0;
      score <= 0;
  end // reset
 else if (win)                  // stay winning
      win <= win;
 else if (lose)                 // stay losing
      lose <= lose;

 else if (score == 63)          // how to win
         win <= 1;
 else if (miss)                 // how to lose
         lose <= 1;
 else  if (catch)               // how to increase score
         score <= score + 1;

 else
  begin
         win <= win;
         lose <= lose;
         score <= score;
  end

endmodule


//  SCORE TEST MODULE
// for testing the score module

`timescale 1 ns / 1 ps

module score_tb ();

reg clock, reset, catch, miss;
wire win, lose;
wire [5:0] score;

initial begin
  clock = 0;
  forever #5 clock = ~clock;
end

initial begin
  reset = 1;
  miss = 0;
  catch = 0;
  #30;             // win=0, lose=0, score=0
  reset = 0;
  catch = 1;
  #50;              // win=0, lose=0, score=1,2,3,4,5
  catch = 0;
  miss = 1;
  #30;              // win=0, lose=1, score=5
// -------------------------------------------------reset----------
  reset = 1;
  miss = 0;
  catch = 0;
  #30;              // win=0, lose=0, score=0
  reset = 0;
  catch = 1;
  #400              // win=0, lose=0, score=growing
  catch = 0;
  miss = 1;
  #30;              // win=0, lose=1, score=???
// -------------------------------------------------reset----------
  reset = 1;
  miss = 0;
  catch = 0;
  #30;              // win=0, lose=0, score=0
  reset = 0;
  catch = 1;
  #650              // win=0, lose=0, score=growing
  #20               // win=1, lose=0, score=63
// -------------------------------------------------reset----------
  reset = 1;
  miss = 0;
  catch = 0;
  #30;              // win=0, lose=0, score=0
  reset = 0;
  catch = 1;
  #650              // win=0, lose=0, score=growing
  #10               // win=1, lose=0, score=63
  catch = 0;
  miss = 1;
  #30;              // win=1, lose=0, score=63
// -------------------------------------------------reset----------
  reset = 1;
  miss = 0;
  catch = 0;
  #30;              // win=0, lose=0, score=0
  reset = 0;
  catch = 1;
  #50               // win=0, lose=0, score=1,2,3...10
  catch = 0;
  miss = 1;
  #30;              // win=0, lose=1, score=10
  miss = 0;
  catch = 1;
  #30;              // win=0, lose=1, score=10

  $stop();
end

score_counter sc(.clock(clock),.reset(reset),
                 .catch(catch),.miss(miss),
                 .win(win),.lose(lose),.score(score));

endmodule
```

Appendix F: pong_game.v

```
 ////////////////////////////////////////////////////////////////////////////////
//
// pong_game: the game itself!
//
////////////////////////////////////////////////////////////////////////////////

module pong_game (
   input vclock,   // 65MHz clock
   input reset,          // 1 to initialize module
   input up,       // 1 when paddle should move up
   input down,     // 1 when paddle should move down
   input [3:0] pspeed,  // puck speed in pixels/tick
   input [10:0] hcount,  // horizontal index of current pixel (0..1023)
   input [9:0]  vcount, // vertical index of current pixel (0..767)
   input hsync,          // XVGA horizontal sync signal (active low)
   input vsync,          // XVGA vertical sync signal (active low)
   input blank,          // XVGA blanking (1 means output black pixel)

   output phsync,  // pong game's horizontal sync
   output pvsync,  // pong game's vertical sync
   output pblank,  // pong game's blanking
   output [23:0] pixel   // pong game's pixel
   output signed [47:0] X0minusXc,
      output signed [47:0] Y0minusYc,
      output signed [47:0] Z0minusZc,
      output reg signed [47:0] X0minusXc_2,
      output reg signed [47:0] Y0minusYc_2,
      output reg signed [47:0] Z0minusZc_2,
      output reg signed [47:0] C,
      output reg signed [47:0] Xd_Sqrd, // 0 to 262_144
      output reg signed [47:0] Yd_Sqrd, // 0 to 147_456
   output reg signed [47:0] Zd_Sqrd, // Always 262_144.
      output reg signed [47:0] XdDotX0minusXc, // 19 bit 2's complement
      output reg signed [47:0] YdDotY0minusYc, // 19 bit 2's complement.
   output reg signed [47:0] ZdDotZ0minusZc, // 21 bit 2's complement.
      output reg signed [47:0] A, // 20 bit integer.
      output reg signed [47:0] B, // 23 bit 2's complement.
      output reg signed [47:0] four_A_C, // 43 bit integer.
      output reg signed [47:0] B_sqrd); // 43 bit integer.

// Module Diagram:
//
// Inputs
//  || (hcount, vcount)
// +-------------+
// |Ray Generator|
// +-------------+
//  |||| (xd[1],yd[1],color[1],t[1])
// +----------------+
// |Puck Intersector| <- puck_x/y/z
// +----------------+
//  |||| (xd[2],yd[2],color[2],t[2])
// +---------------------+
// |Left Wall Intersector|
// +---------------------+
//  |||| (xd[3],yd[3],color[3],t[3])
// +----------------------+
// |Right Wall Intersector|
// +----------------------+
//  |||| (xd[4],yd[4],color[4],t[4])
// +-----------------+
// |Floor Intersector|
// +-----------------+
//  |||| (xd[5],yd[5],color[5],t[5])
// +-------------------+
// |Ceiling Intersector|
// +-------------------+
//  |||| (xd[6],yd[6],color[6],t[6])
// +------------------+
// |Paddle Intersector|
// +------------------+
//  |||| (xd[7],yd[7],color[7],t[7])
// +-------------+
// |Color Manager| <- puck_x/y/z, paddle_x/y
// +-------------+
//  |||
// Outputs



      reg signed [9:0]   puck_x; // -472 to 472
      reg signed [9:0]   puck_y; // -344 to 344
      reg signed [10:0]  puck_z; //   40 to 1023
      reg signed [9:0] paddle_x; // -412 to 412
      reg signed [9:0] paddle_y; // -284 to 284
      reg [24:0] counter;



      always @(posedge vclock) begin
            if (reset) begin
                  puck_x <= 100; // Placeholders
                  puck_y <= 0; // for
                  puck_z <= 620; // testing

                  paddle_x <= -40;
                  paddle_y <= 40;
                  counter <= 0;
            end else if (counter >= 25'b00_0001_1001_1011_1111_1100_110) begin
                  puck_x <= puck_x; // Placeholders
                  puck_y <= puck_y; // for
                  puck_z <= puck_z; // testing

                  paddle_x <= paddle_x;
                  paddle_y <= paddle_y;
                  counter <= 0;
            end else if (up) begin
                  counter <= counter + 1;
            end
      end


      wire signed [9:0] xd[7:1];  // -512 to 511
      wire signed [9:0] yd[7:1];  // -512 to 511
      wire [1:0] color[7:1];      // 0 to 3
      wire [13:0] t[7:1];

      wire [7:0] red;
      wire [7:0] green;
      wire [7:0] blue;

      // RAY GENERATOR
      ray_generator ray_gen(.clk(vclock), .reset(reset),
       // Input
       .hcount(hcount), .vcount(vcount),
       // Output
              .rayX(xd[1]),
              .rayY(yd[1]),
          .color(color[1]),
                  .tOut(t[1]));

      // PUCK
      fast_sphere_intersector puck(.reset(reset), .clk(vclock),
       // Sphere coordinates.
       .Xc(puck_x), .Yc(puck_y), .Zc(puck_z),
       // Ray coordinates input.
                .Xd(xd[1]),
                          .Yd(yd[1]),
        .colorIn(color[1]),
                .tIn(t[1]),
       // Ray coordinates output,
             .outX(xd[2]),
             .outY(yd[2]),
       .colorOut(color[2]),
                         .tOut(t[2]));

      // LEFT WALL ** complete **
      plane_intersector  #(.X_NORMAL(1),.Y_NORMAL(0),.Z_NORMAL(0),.DISTANCE(-
320),.COLOR(1),.X_MAX(-163))
      left_wall (
            // input
            .reset(reset), .clk(vclock),
                    .Xd(xd[2]),
                    .Yd(yd[2]),
            .colorIn(color[2]),
                    .tIn(t[2]),
            // output
                  .outX(xd[3]),
                  .outY(yd[3]),
        .colorOut(color[3]),
                   .tOut(t[3]));

      // RIGHT WALL ** complete **
      plane_intersector  #(.X_NORMAL(-1),.Y_NORMAL(0),.Z_NORMAL(0),.DISTANCE(-
320),.COLOR(1),.X_MIN(162))
      right_wall (
            // input
            .reset(reset), .clk(vclock),
                    .Xd(xd[3]),
                           .Yd(yd[3]),
            .colorIn(color[3]),
                    .tIn(t[3]),
            // output
                  .outX(xd[4]),
                         .outY(yd[4]),
        .colorOut(color[4]),
                   .tOut(t[4]));

      // FLOOR ** complete **
      plane_intersector  #(.X_NORMAL(0),.Y_NORMAL(-
1),.Z_NORMAL(0),.DISTANCE(240),.COLOR(1),.Y_MAX(-120))
      floor (
            // input
            .reset(reset), .clk(vclock),
                    .Xd(xd[4]),
                           .Yd(yd[4]),
            .colorIn(color[4]),
                    .tIn(t[4]),
            // output
                  .outX(xd[5]),
                         .outY(yd[5]),
        .colorOut(color[5]),
                .tOut(t[5]));

      // CEILING ** complete **
      plane_intersector
      #(.X_NORMAL(0),.Y_NORMAL(1),.Z_NORMAL(0),.DISTANCE(240),.COLOR(1),.Y_MIN(11
9))
      ceiling (
            // input
            .reset(reset), .clk(vclock),
                     .Xd(xd[5]),
                           .Yd(yd[5]),
             .colorIn(color[5]),
                     .tIn(t[5]),
            // output
                   .outX(xd[6]),
                          .outY(yd[6]),
            .colorOut(color[6]),
                    .tOut(t[6]));

      // PADDLE ** complete **
      plane_intersector  #(.X_NORMAL(0),.Y_NORMAL(0),.Z_NORMAL(-
1),.DISTANCE(640),.COLOR(3))
      paddle_plane (
            // input
            .reset(reset), .clk(vclock),
                     .Xd(xd[6]),
                           .Yd(yd[6]),
             .colorIn(color[6]),
                     .tIn(t[6]),
            // output
                   .outX(xd[7]),
                          .outY(yd[7]),
            .colorOut(color[7]),
                    .tOut(t[7]));

      // COLOR MANAGER
      good_color_manager colormanager(
            // input
            .reset(reset), .clk(vclock),
                   .inX(xd[7]),
                          .inY(yd[7]),
            .colorIn(color[7]),
                    .tIn(t[7]),
            .puck_x(puck_x),.puck_y(puck_y),.puck_z(puck_z),
            .paddle_x(paddle_x),.paddle_y(paddle_y),
            // output
            .red(red),.green(green),.blue(blue));

  assign pixel = {red,green,blue};

      // Pipeline the VGA signals.
      pipe hsync_pipe(.clk(vclock), .in(hsync), .out(phsync));
      pipe vsync_pipe(.clk(vclock), .in(vsync), .out(pvsync));
      pipe blank_pipe(.clk(vclock), .in(blank), .out(pblank));

endmodule
```

Appendix G: ray_generator.v

```
module ray_generator(input clk,
                                         input reset,
                                            input [10:0] hcount,     //
horizontal index of current pixel (0..1023)
                                            input [9:0]  vcount, // vertical
index of current pixel (0..767)
                                            output reg signed [9:0] rayX,
                                            output reg signed [9:0] rayY,
                                            output reg [1:0] color,
                                            output reg [13:0] tOut);

      // Colors:
      // 0: None decided yet.
      // 1: Wall.
      // 2: Puck.
      // 3: Paddle.

      // T is a value ranging from 0 to 3 + 4095/4096
      // - long enough to reach anywhere in the box.
      // Ray origin: X=0, Y=0, Z=-512.
      // Farthest possible ray destination: X=-512, Y=-384, Z=1023.
      // Distance: ~1662, = ~3.2 * 511, definitely reachable.

      always @(posedge clk) begin
            if (reset) begin
                  rayX <= 0;
                  rayY <= 0;
                  color <= 0;
                  tOut <= 14'b11_1111_1111_1111;
            end else begin
                  rayX <= -320 + hcount;
                  rayY <=  239 - vcount;
                  color <= 0;
                  tOut <= 14'b11_1111_1111_1111;
            end
      end
endmodule
```

Appendix H: fast_sphere_intersector.v

```
module fast_sphere_intersector(
      input reset,
      input clk,
      // Sphere coordinates.
      //  Xc: -472 <= val <=  472
      //  Yc: -344 <= val <=  344
      //  Zc:   40 <= val <= 1023
      input signed [9:0] Xc,
      input signed [9:0] Yc,
      input signed [10:0] Zc,
      // Ray coordinates
      //  X0: 0.
      //  Y0: 0.
      //  Z0: -512.
      //  Xd: -512 to +511.
      //  Yd: -384 to +383.
      //  Zd: +512, thus Zd^2 = 262_144.
      input signed [9:0] Xd,
      input signed [9:0] Yd,
      // Previously calculated values;
      // may be replaced, might not.
      input [1:0] colorIn,
      input [13:0] tIn,
      // Outputted valued include:
      //  outX and outY, the Xd and Yd
      //  colorOut, the new color
      //  tOut, the intersection of that ray.
      output reg signed [9:0] outX,
      output reg signed [9:0] outY,
      output reg [1:0] colorOut,
      output reg [13:0] tOut,
      // Testing output ports
      output signed [47:0] X0minusXc,
      output signed [47:0] Y0minusYc,
      output signed [47:0] Z0minusZc,
      output reg signed [47:0] X0minusXc_2,
      output reg signed [47:0] Y0minusYc_2,
      output reg signed [47:0] Z0minusZc_2,
      output reg signed [47:0] C,
      output reg signed [47:0] Xd_Sqrd, // 0 to 262_144
      output reg signed [47:0] Yd_Sqrd, // 0 to 147_456
   output reg signed [47:0] Zd_Sqrd, // Always 262_144.
      output reg signed [47:0] XdDotX0minusXc, // 19 bit 2's complement
      output reg signed [47:0] YdDotY0minusYc, // 19 bit 2's complement.
   output reg signed [47:0] ZdDotZ0minusZc, // 21 bit 2's complement.
      output reg signed [47:0] A, // 20 bit integer.
      output reg signed [47:0] B, // 23 bit 2's complement.
      output reg signed [47:0] four_A_C, // 43 bit integer.
      output reg signed [47:0] B_sqrd); // 43 bit integer.

// The fast sphere intersector checks whether intersection occurs,
// but it doesn't calculate WHERE it occurs. Thus it's only useful
// for flat-shading and only if we have a pre-existing rendering order.
// For example, in our application, the sphere always takes priority,
// so we just replace whatever else the ray might have with the sphere's color.

integer i;

// Constants:
//  X0 and Y0: 0. Just factor these out of the equations.
//  Z0: -512.
//  Xd: -512 to +511.
//  Yd: -384 to +383.
//  Zd: +512, thus Zd^2 = 262_144.
//  Xc: -472 <= val <=  472
//  Yc: -344 <= val <=  344
//  Zc:   40 <= val <= 1023
//  Sr: +40, thus Sr^2 = 1600.

wire signed [9:0] Z0;
wire signed [9:0] Zd;
assign Z0 = -320;
assign Zd = 320;

// SPHERE-COORDINATE & CONSTANT BASED CALCS
// X/Y/Z0 - X/Y/Zc
// (X0-Xc) = -Xc.
//  bits: 10 bit 2's complement.
// (Y0-Yc) = -Xc
//  bits: 10 bit 2's complement.
// (Z0-Zc) = -552 to -1535
//  bits: 12 bit 2's complement.

//    wire signed  [9:0] X0minusXc;
//    wire signed  [9:0] Y0minusYc;
//    wire signed [11:0] Z0minusZc;
      assign X0minusXc =  0 - Xc;
      assign Y0minusYc =  0 - Yc;
      assign Z0minusZc = Z0 - Zc;

// (X/Y/Z0-X/Y/Zc)^2
// X: min:       0. max:   222_784.
// Y: min:       0. max:   118_336.
// Z: min: 304_704. max: 2_356_225.

//    wire [17:0] X0minusXc_2;
//    wire [16:0] Y0minusYc_2;
//    wire [21:0] Z0minusZc_2;
//    assign X0minusXc_2 = X0minusXc * X0minusXc;
//    assign Y0minusYc_2 = Y0minusYc * Y0minusYc;
//    assign Z0minusZc_2 = Z0minusZc * Z0minusZc;

      // C = Xc^2 + Yc^2 + (-512-Zc)^2 - Sr^2
      //  min of C = 0 + 0 + 304_704 - 1600
      //                      = 303_104
      //  max of C = 222_784 + 118_336 + 2_356_225 - 1600
      //                      = 2_695_745
      //  bits for C: 22 bit integer.
//    wire [21:0] C;
      // assign C = X0minusXc_2 + Y0minusYc_2 + Z0minusZc_2 - 1600;

// REGISTERS FOR PIPELINE PHASES:
// # Multi-Phase Registers:
//  # Phase 1-3

      reg signed [9:0] regX[3:1];
      reg signed [9:0] regY[3:1];
      reg [1:0] regColor[3:1];
      reg [13:0] regT[3:1];

// # Phase 1:
//  * Calculate X/Y/Zd^2
//  * Calculate Yd^2
//  * We already know Zd^2 = 262_144.
//  * Calculate Xd*(X0-Xc) and for Y and Z.

//    reg [18:0] Xd_Sqrd; // 0 to 262_144
//    reg [17:0] Yd_Sqrd; // 0 to 147_456
//wire [18:0] Zd_Sqrd; // Always 262_144.
   //assign Zd_Sqrd = 19'd262_144;

// Xd*(X0-Xc)
//  min: -241_664. Max: 241_664.
// Yd*(Y0-Yc)
//  min: -132_096. Max: 132_096.
// Zd*(Z0-Zc) = 512*(-552 to -1535)
//  min: -785_920. Max: -282_624.
//    reg signed [18:0] XdDotX0minusXc; // 19 bit 2's complement
//    reg signed [18:0] YdDotY0minusYc; // 19 bit 2's complement.
//wire signed [20:0] ZdDotZ0minusZc; // 21 bit 2's complement.
      // assign ZdDotZ0minusZc = 512 * Z0minusZc;

// # Phase 2:
//  * From X/Y/Zd^2, calculate A.
//  * From X/Y/Zd*(X/Y/Z0-X/Y/Zc), calculate B.
//  * We already know C.

// A = Xd^2 + Yd^2 + Zd^2
//  min = 261_121. max = 670_721.
//    reg [19:0] A; // 20 bit integer.
// B = 2 * (-Xd*Xc - Yd*Yc + Zd*(Z0-Zc))
//  min = -2_319_360. max = 182_272.
//    reg signed [22:0] B; // 23 bit 2's complement.

// # Phase 3:
//  * From A and C, calculate 4*A*C.
//  * From B, calculate B^2.

// 4AC
//  min =   316_587_278_336
//  max = 7_232_371_128_580
//    reg [42:0] four_A_C; // 43 bit integer.

// B^2
//  min = 0. max = 5_379_430_809_600.
//    reg [42:0] B_sqrd; // 43 bit integer.

// # Phase 4 (OUTPUT):
//  * If 4AC > B^2, no intersection.
//   @ Pass on colorIn as colorOut, tIn as tOut
//  * Otherwise, intersection.
//   @ Replace colorOut with 2'b10.
//   @ Replace tOut with 1.

      always @(posedge clk) begin

            // Phase 1-3
                regX[1]    <= Xd;
                regY[1]    <= Yd;
            regColor[1]    <= colorIn;
                regT[1]    <= tIn;
            for (i = 1; i <3; i = i+1) begin
                   regX[i+1] <=     regX[i];
                regY[i+1] <=     regY[i];
            regColor[i+1] <= regColor[i];
                regT[i+1] <=     regT[i];
            end

            // Phase 1
            Xd_Sqrd        <= Xd*Xd;
            Yd_Sqrd        <= Yd*Yd;
            Zd_Sqrd                  <= Zd*Zd;
            XdDotX0minusXc <= Xd*X0minusXc;
            YdDotY0minusYc <= Yd*Y0minusYc;
            ZdDotZ0minusZc <= Zd*Z0minusZc;
            X0minusXc_2        <= X0minusXc*X0minusXc;
            Y0minusYc_2        <= Y0minusYc*Y0minusYc;
            Z0minusZc_2        <= Z0minusZc*Z0minusZc;

            // Phase 2
            A                              <= Xd_Sqrd + Yd_Sqrd + Zd_Sqrd;
            B                              <= (2 * XdDotX0minusXc) + (2 *
YdDotY0minusYc) + (2 * ZdDotZ0minusZc);
            C                              <= X0minusXc_2 + Y0minusYc_2 +
Z0minusZc_2 - 40000;

            // Phase 3
            four_A_C           <= 4 * A * C;
            B_sqrd                   <= B*B;

            // Phase 4
            outX                     <=     regX[3];
            outY                     <=     regY[3];
            if (four_A_C > B_sqrd) begin
                  // No intersection.
              colorOut         <= regColor[3];
                  tOut         <=     regT[3];
            end else begin
                  // Intersection.
              colorOut         <= 2'd2;
                  tOut         <= 14'b00_1111_1111_1111;
            end
      end
endmodule
```

Appendix I: plane_intersector.v

```
      // complete

module plane_intersector
      #(parameter signed [2:0] X_NORMAL = 0,
        parameter signed [2:0] Y_NORMAL = 0,
        parameter signed [2:0] Z_NORMAL = 0,
        parameter signed [9:0] DISTANCE = 0,
        parameter [2:0] COLOR = 2'b01,
        parameter signed [9:0] X_MIN = -320,
        parameter signed [9:0] X_MAX = 319,
        parameter signed [9:0] Y_MIN = -240,
        parameter signed [9:0] Y_MAX = 239) (
      input reset,
      input clk,
      // Ray coordinates
      //  X0: 0.
      //  Y0: 0.
      //  Z0: -512.
      //  Xd: -512 to +511.
      //  Yd: -384 to +383.
      //  Zd: +512, thus Zd^2 = 262_144.
      input signed [9:0] Xd,
      input signed [9:0] Yd,
      // Previously calculated values;
      // may be replaced, might not.
      input [1:0] colorIn,
      input [13:0] tIn,
      // Outputted valued include:
      //  outX and outY, the Xd and Yd
      //  colorOut, the new color
      //  tOut, the intersection of that ray.
      output reg signed [9:0] outX,
      output reg signed [9:0] outY,
      output reg [1:0] colorOut,
      output reg [13:0] tOut);

// For-Loop Master
integer i;

// The plane-intersector calculates IF and WHERE a ray intersects a plane.

// CONSTANTS:
//  X0 and Y0: 0. Just factor these out of the equations.
//  Z0: -320.
//  Xd: -320 to +319.
//  Yd: -240 to +239.
//  Zd: +320, thus Zd^2 = 262_144.
//  Xn, Yn, Zn, D: Parameterized.

// PLANE-COORDINATE & CONSTANT BASED CALCS

// A*X0 = 0.
// B*Y0 = 0.
// C*Z0 = needs actual calculation.

      wire signed [10:0] Zn_Z0 = Z_NORMAL * -320;

// REGISTERS FOR PIPELINE PHASES:
// # Multi-Phase Registers:
//  # Phase 1-27

      reg signed [9:0] regX[27:1];
      reg signed [9:0] regY[27:1];
      reg         [1:0] regC[27:1];
      reg        [13:0] regT[27:1];
```

```
// PHASE 1:

// X/Y/Zn*X/Y/Zd
// x min -512, max 512
// y min -384, max 384
// z min -512, max 512
      reg signed [10:0] Xn_Xd;
      reg signed [10:0] Yn_Yd;
      wire signed [10:0] Zn_Zd;
      assign Zn_Zd = Z_NORMAL * 320;

// PHASE 2:

// Pn*R0+D
//  min -512, max 1535
      reg signed [10:0] Pn_R0_D_magnitude;
      reg Pn_R0_D_sign; // 12-bit Sign-magnitude.

// Pn*Rd
//  min -512, max 512
      reg signed [9:0] Pn_Rd_magnitude;
      reg Pn_Rd_sign; // 11-bit Sign-magnitude.

// PHASE 3-27:
      reg intersect[27:3]; // Did this line intersect at all?

// PHASE 27:
      reg [23:0] tLocal;

      wire [23:0] dividerOut;

// Divider Module
      divide_23_bits_by_10_bits plane_divider(.clk(clk),
            // Input
            .dividend({Pn_R0_D_magnitude,12'h000}),
            .divisor(Pn_Rd_magnitude),
            // Output
            .quotient(dividerOut),
            // Unused Output
            .remainder(), .rfd());

      always @(posedge clk) begin
            // PHASE 1-27

            regX[1] <= Xd;
            regY[1] <= Yd;
            regC[1] <= colorIn;
            regT[1] <= tIn;

            for (i = 1; i<27;i=i+1) begin
                  regX[i+1] <= regX[i];
                  regY[i+1] <= regY[i];
                  regC[i+1] <= regC[i];
                  regT[i+1] <= regT[i];
            end

            outX <= regX[27];
            outY <= regY[27];

            // Phase 1

            Xn_Xd = X_NORMAL * Xd;
            Yn_Yd = Y_NORMAL * Yd;

            // Phase 2

            // Pn*R0+D
            if (Z_NORMAL * -320 + DISTANCE < 0) begin
            Pn_R0_D_magnitude <= Z_NORMAL * 512 - DISTANCE;
            Pn_R0_D_sign <= 1;
         end else begin
            Pn_R0_D_magnitude <= Z_NORMAL * -512 + DISTANCE;
            Pn_R0_D_sign <= 0;
         end

         // Pn*Rd
         if (Xn_Xd + Yn_Yd + Zn_Zd < 0) begin
            Pn_Rd_magnitude <= -1*(Xn_Xd + Yn_Yd + Zn_Zd);
            Pn_Rd_sign <= 1;
         end else begin
            Pn_Rd_magnitude <= Xn_Xd + Yn_Yd + Zn_Zd;
            Pn_Rd_sign <= 0;
         end

         // Phase 3-27
         if ((Pn_Rd_sign != Pn_R0_D_sign) || (Pn_R0_D_magnitude == 0)) begin
            // If the plane is behind or parallel to the ray,
            // We have no intersection.
            intersect[3] <= 1'b0;
         end else begin
            // Otherwise we have an intersection.
            intersect[3] <= 1'b1;
         end

         for (i=3; i<27;i=i+1) begin
            intersect[i+1] <= intersect[i];
         end

         // Phase 27

         tLocal <= dividerOut;

         // Phase 28
         if (intersect[27] && (tLocal[13:0] < regT[27]) && (regX[27] >= X_MIN)
&& (regX[27] <= X_MAX) && (regY[27] >= Y_MIN) && (regY[27] <= Y_MAX)) begin
            // If there's an intersection,
            // and the intersection is before other intersections,
            // and we're in acceptable x and y values
            // change the t and color.
           colorOut <= COLOR;
               tOut <= tLocal[13:0];
         end else begin
            // Leave the t and color alone.
           colorOut <= regC[27];
               tOut <= regT[27];
         end


      end
endmodule
```

Appendix J: good_color_manager.v

```
module good_color_manager(
      input clk,
      input reset,
      input signed [9:0] inX,
      input signed [9:0] inY,
      input signed [9:0] puck_x,
      input signed [9:0] puck_y,
      input signed [10:0] puck_z,
      input signed [9:0] paddle_x,
      input signed [9:0] paddle_y,
      input [1:0] colorIn,
      input [13:0] tIn,
      output reg [7:0] red,
      output reg [7:0] green,
      output reg [7:0] blue);

      reg signed [22:0] point_x;
      reg signed [22:0] point_y;
      reg signed [23:0] point_z;
      reg [1:0] point_c;
      reg [7:0] point_t;
   reg signed [10:0] point2_x;
      reg signed [10:0] point2_y;
      reg signed [11:0] point2_z;
      reg [1:0] point2_c;
      reg [7:0] point2_t;

      wire signed [10:0] puck2_x;
      wire signed [10:0] puck2_y;
      wire signed [10:0] paddle2_x;
      wire signed [10:0] paddle2_y;

      assign puck2_x = 320 + puck_x;
      assign puck2_y = 240 + puck_y;

      assign paddle2_x = 320 + paddle_x;
      assign paddle2_y = 240 + paddle_y;

      always @(posedge clk) begin
            // Phase 1
            point_x = (320*14'b01_0000_0000_0000) + (tIn * inX);
            point_y = (240*14'b01_0000_0000_0000) + (tIn * inY);
            point_z = (tIn - 14'b01_0000_0000_0000) * 320;
            point_t = tIn[13:6];
            point_c = colorIn;

            // All x and Y values positive'd with 0,0 in the lower left.

            // Phase 2
            point2_x = point_x >>> 12;
            point2_y = point_y >>> 12;
            point2_z = point_z >>> 12;
            point2_t = point_t;
            point2_c = point_c;

            // All x, y, and z values rounded down to integer.

            // Phase 3
            if (point2_c == 2'b11) begin
                  // paddle
                  if (((point2_x < puck_x + 50 ) && (point2_x > puck_x - 50 )) &&
((point2_y < puck_y + 50 ) && (point2_y > puck_y - 50 ))) begin
                        red   <= 8'hFF;
                        green <= 8'hFF;
                        blue  <= 8'hFF;
                  end else begin
```

```
                        red   <= 0;
                        green <= 0;
                        blue  <= 0;
                end
        end else if (point2_c == 2'b10) begin
                // puck
                red   <= point2_t;
                green <= 0;
                blue  <= 0;
        end else if (point2_c == 2'b01) begin
                // wall
                case({point_x[18],point_y[18],point_z[18]})
                        3'b000: {red,green,blue} <= 24'h00_00_00;
                        3'b001: {red,green,blue} <= 24'h00_00_FF;
                        3'b010: {red,green,blue} <= 24'h00_FF_00;
                        3'b011: {red,green,blue} <= 24'h00_FF_FF;
                        3'b100: {red,green,blue} <= 24'hFF_00_00;
                        3'b101: {red,green,blue} <= 24'hFF_00_FF;
                        3'b110: {red,green,blue} <= 24'hFF_FF_00;
                        3'b111: {red,green,blue} <= 24'hFF_FF_FF;
                endcase
        end else begin
                {red,green,blue} <= {point2_t,point2_t,{8'h00}};
        end
    end
endmodule
```