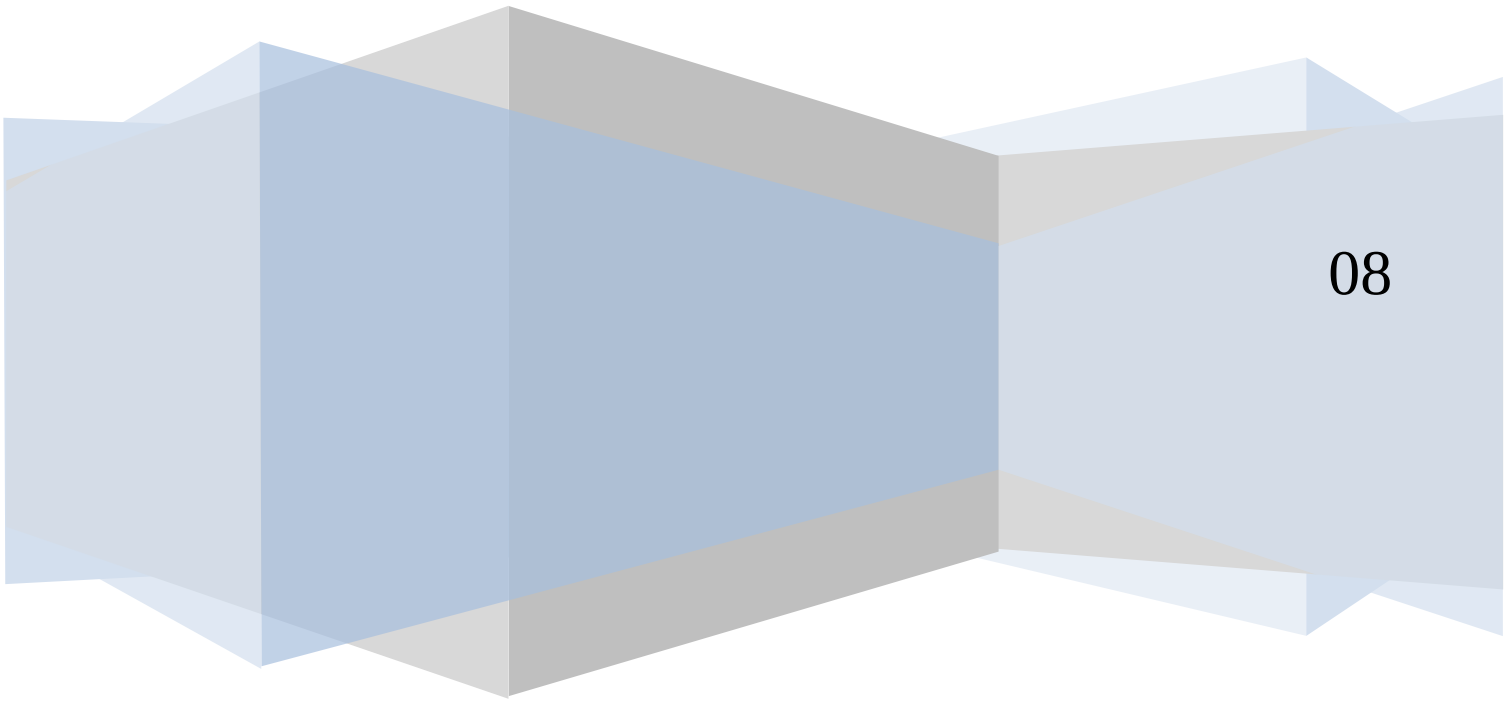
Design Document

A Simple, real time hardware ray tracer

David Smith, Daniel Benamy, Keerti Joshi, Minjie Zhang



08

Milestones

1: All computations / low level modules are working
2: The system is integrated and we can write to our memory from a C program on the NIOS.
3: Low resolution video

Final: An improved resolution video and we have a simple video game or simulation running from the NIOS processor.

Hardware Module Hierarchy

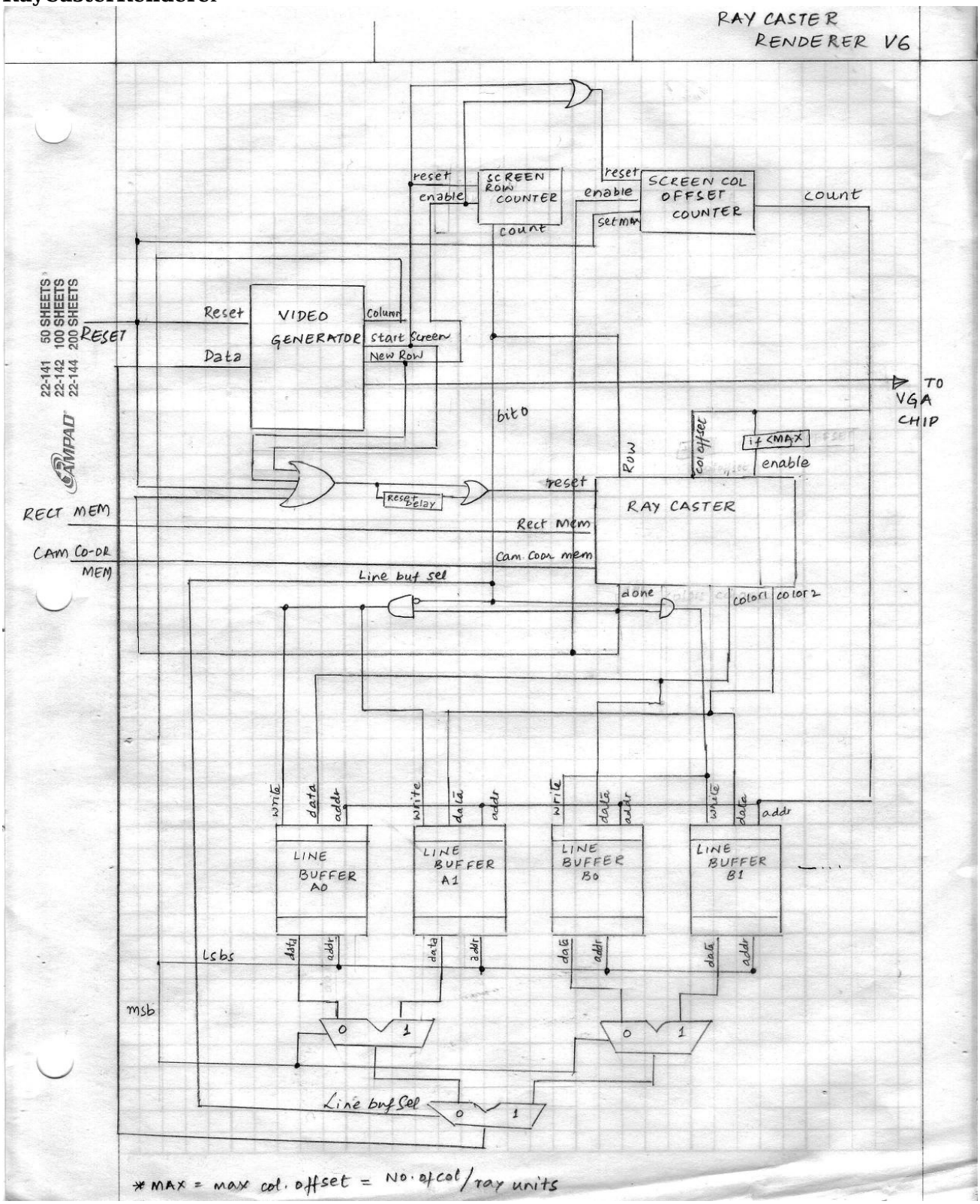
RayCasterProject - connects pins
- RayCasterSystem - built by SOPC Builder
  - NIOS Processor
  - Avalon Bus Stuff
  - SRAM Interface (other generated stuff)
  - JTAG Debug Interface
  - RayCasterModule - our stuff
    - RayCasterRenderer
    - RectMem - stores world rectangles
    - CameraMem - stores camera position
    - AvalonInterface - allows access to our memories over the bus

Hardware Modules

The diagrams we've produced are somewhere between block diagrams and logic diagrams. They're closer to the latter, but are a little handwavy about certain aspect such as how may bits signals are and some details about memory accesses. As many important issues are fleshed out as we could get to.

Rays will be represented by 6 numbers: X, Y, Z, DX, DY, and DZ. The D variables are the slopes. Coordinates in our world space and slopes will be represented by 9 bits.

RayCasterRenderer



At a high level there are three systems here. There's the ray casting system which is composed of the RayCaster module and the screen row and screen column counters. There's the video generator which takes the data produced by the RayCaster and sends it

out to the VGA chip. And there's a double buffered line buffer which passes data between the two.

The video generator is the main driver of the whole system. When it's about to start drawing the screen, it asserts the StartScreen signal. This resets both counters and the RayCaster. The RayCaster will wake up, take the information from the row, column offset, RectMem, and CamCoordMem lines and will start processing.

The design that we drew has two ray units computing in parallel. When we build this, we plan on having 32 ray units in parallel. The design scales in an obvious way but we didn't want to make a big mess on the paper. For the diagram shown, if we'll be working with the resolution 320 by 200, the max constant which appears in a couple of places would be 160, which is 320 columns / 2 parallel ray units. For our final product, the max constant will probably be 10, which is 320 columns / 32 parallel units.

The line buffer is broken up in two ways. The first way is that there's an A buffer and a B buffer. If the RayCaster is writing to the A buffer the VideoGenerator will be reading from the B buffer and vice versa. This is controlled using bit zero of the screen row counter which we call LineBufSel. The second way that the line buffer is broken up is that there's a separate section for each chunk of pixels that we do in parallel. This allows us to write them all at once. So for a row of 320 pixels, the 0 buffers will contain pixels 0 through 159 and the 1 buffers will contain pixels 160 through 319.

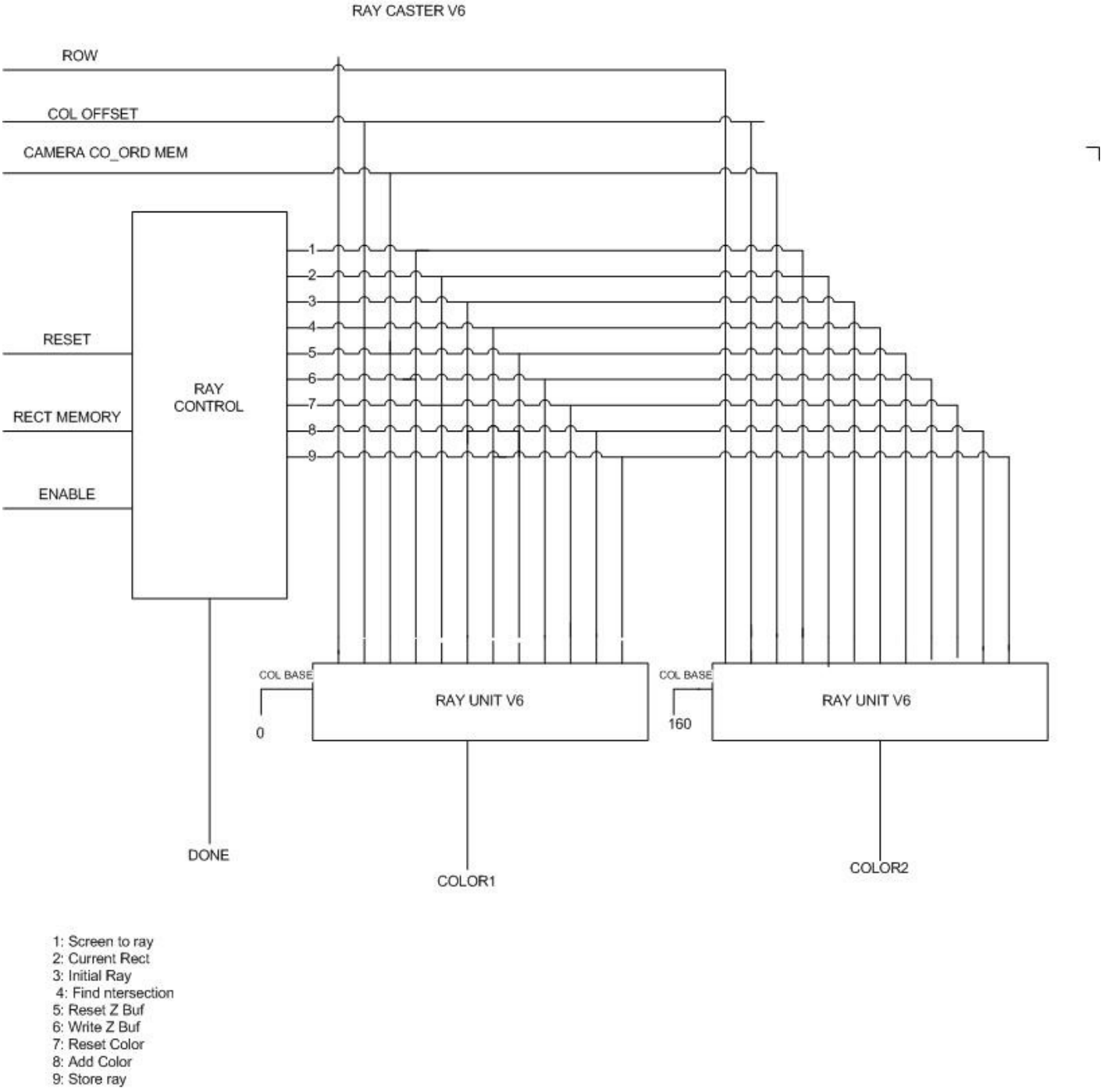
When the RayCaster finishes producing the colors on its output lines, it will bring the done signal high. This will cause its outputs to be written to the appropriate parts of the line buffer, the column offset counter to be incremented, and itself to be reset. It will then start again on the next columns.

With what we've described so far, after the column offset counter reached the maximum value, the RayCaster would continue to compute values and start overwriting memory that we want. To prevent this, when that counter reaches the Max value (which is one more than the last value we compute) we'll stop the RayCaster from running again with the use of the comparator between the column offset and the Max value feeding into the RayCaster's enable line. This will also stop the column counter and prevent it from rolling over and having the enable line be true again.

When the column offset counter changes and the RayCaster is reset, we don't want the RayCaster to start immediately because we need to wait for the comparator on its enable line to produce the correct output. So we've added the ResetDelay flip flop and the or gate in front of its reset line. The RayCaster's reset is synchronous. If any of the lines feeding into its reset are asserted, on the next cycle the RayCaster will reset. Additionally on that second cycle the reset delay flip flop will grab that reset value and hold it for the following cycle. This will cause the reset line to be asserted for two cycles. This is illustrated in the accompanying timing diagrams.

The column line coming out of the video generator represents the column that it needs to read for video output and is independent of the column counter that we use for generating rays.

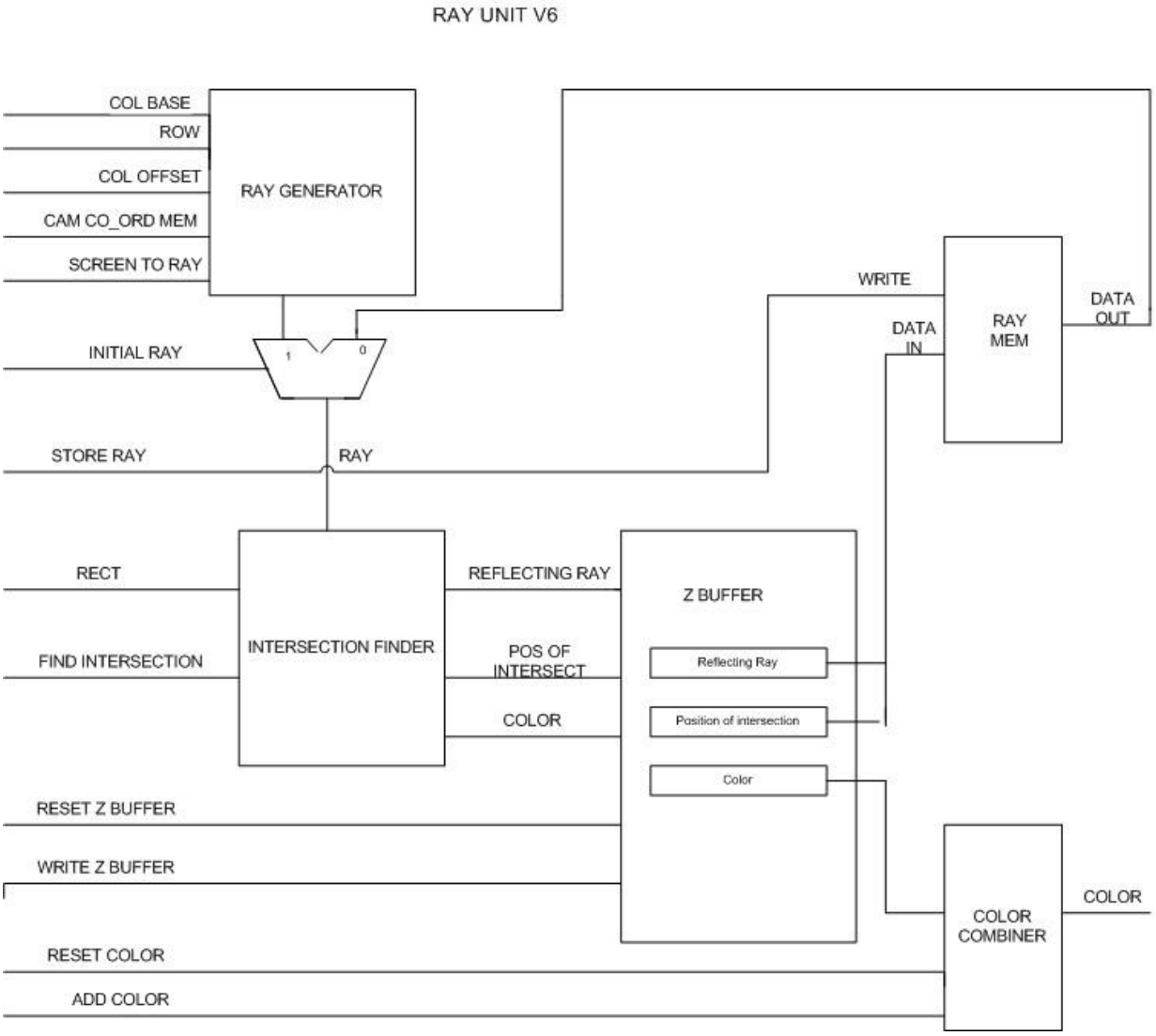
Ray Caster



1: Screen to ray
2: Current Rect
3: Initial Ray
4: Find ntersection
5: Reset Z Buf
6: Write Z Buf
7: Reset Color
8: Add Color
9: Store ray

The RayControl module is intimately connected with the RayUnits. We only have one of them in the RayCaster because it will control many ray units running in parallel. One of the jobs of the ray control is to cycle through all the rectanglels in the RectMem and present the current one on the CurrentRect lines. Other than that it will assert the various signals in the proper order at the proper times in order to operate the ray units.

RayUnit



To start off, ResetZBuffer and ResetColor will be asserted, which will clear the ZBuffer and ColorCombiner.

Initially, Row, ColumnBase, ColumnOffset, and CameraCoordMem will have values on them. The ScreenToRay signal will be asserted and the RayGenerator will produce X, Y, Z and DX, DY, and DZ values. The InitialRay signal will be 1 so the values from the RayGenerator will be directed to the IntersectionFinder. The first rectangle will be presented on the Rect lines and FindIntersection will be asserted. Some serious math will happen and then the IntersectionFinder will produce the X, Y, and Z coordinates of the intersection, DX, DY, and DZ of the reflected ray, and the color at that intersection. The WriteZBuf line will be asserted and the Z Buffer will decide whether those values are closer to the source than the previous values and if so, store those new values.

The FindIntersection and WriteZBuf steps will be repeated for every rectangle. Each time a new rectangle will be on the Rect lines. After all of them have been processed, AddColor will be asserted which will cause the ColorCombiner to grab the resulting color from the Z Buffer. StoreRay will also be asserted which will cause RayMem to store the values of the reflecting ray from the Z Buffer.

The entire process will be repeated to handle the reflection except that InitialRay will be 0 so we'll be finding the intersection from the reflected ray instead of from the initial camera ray. This time when the ColorCombiner adds the color, it will combine the previous color and the new color to produce a color value that includes the reflection.

Timing

Cycle Budget

Rows	Cols	Pixels / Frame	FPS	Pixels / Sec	# of Ray Units	Pixels / Sec / Ray Unit	Sec / Pixel / Ray Unit	Clock Freq	Clock Period	Cycles / Pixel / Ray Unit
200	320	64,000	60	3,840,000	32	120,000	8.3333E-06	50MHz	2E-08	417

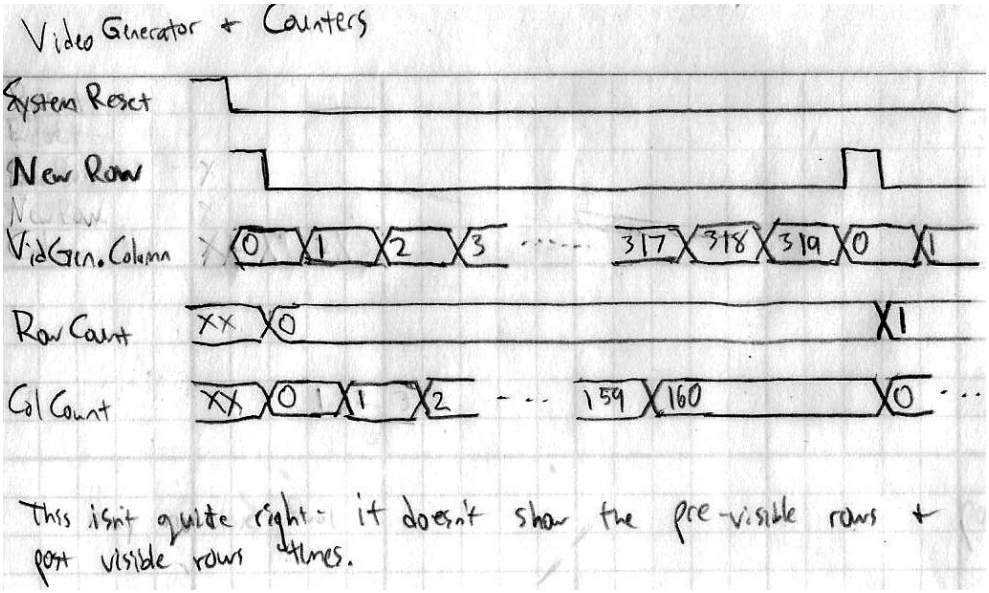
The table illustrates the number of cycles we have for each ray unit to process 1 pixel.

Cycles Required

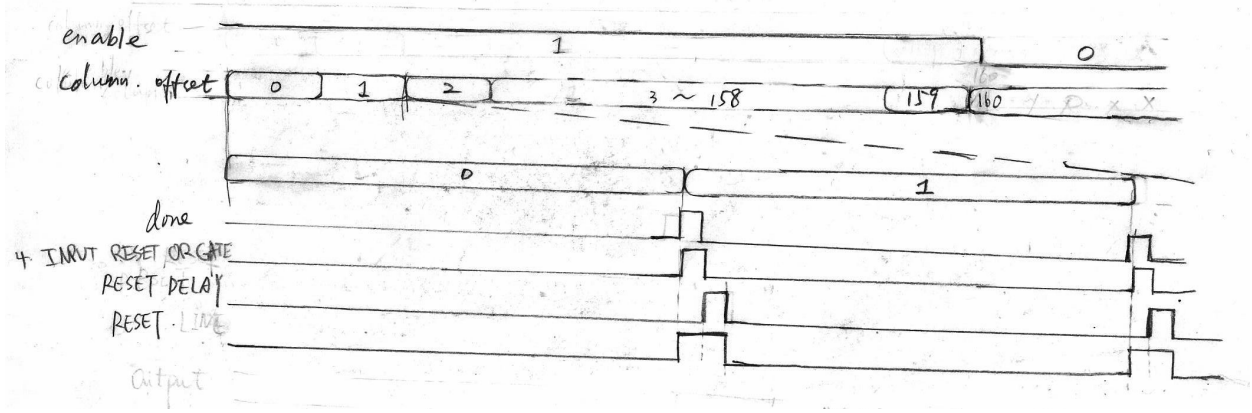
Scene Rectangles	Reflections	Rays / Pixel	Cycles to Divide	Cycles to Multiply	Intersection Helper	Cycles / Ray / Rectangle	Cycles / Pixel
12	1	2	9	1	13	17	408

Timing Diagrams

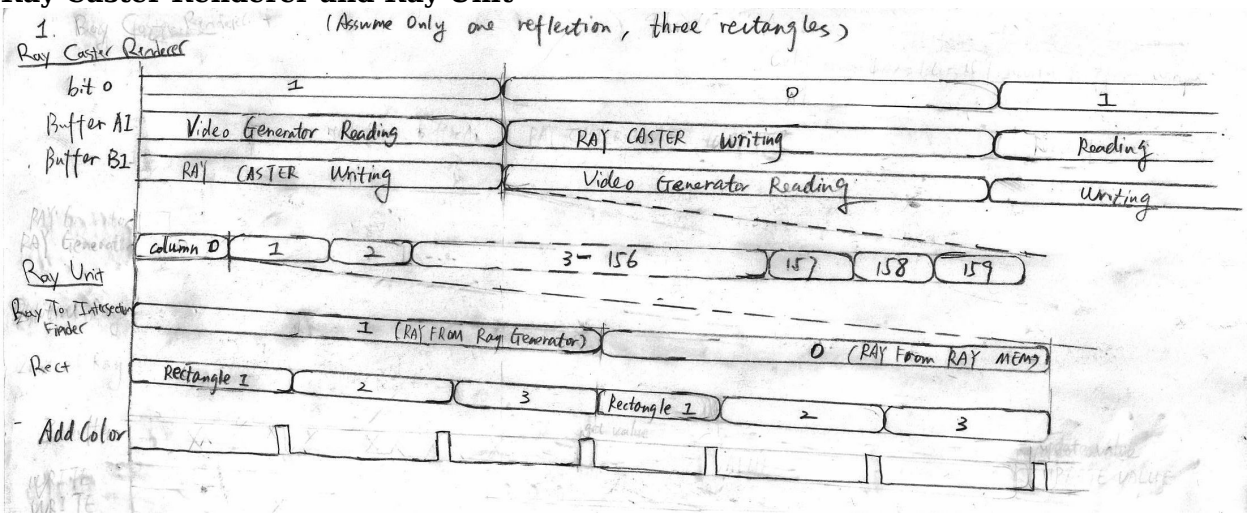
Video Generator and Counters



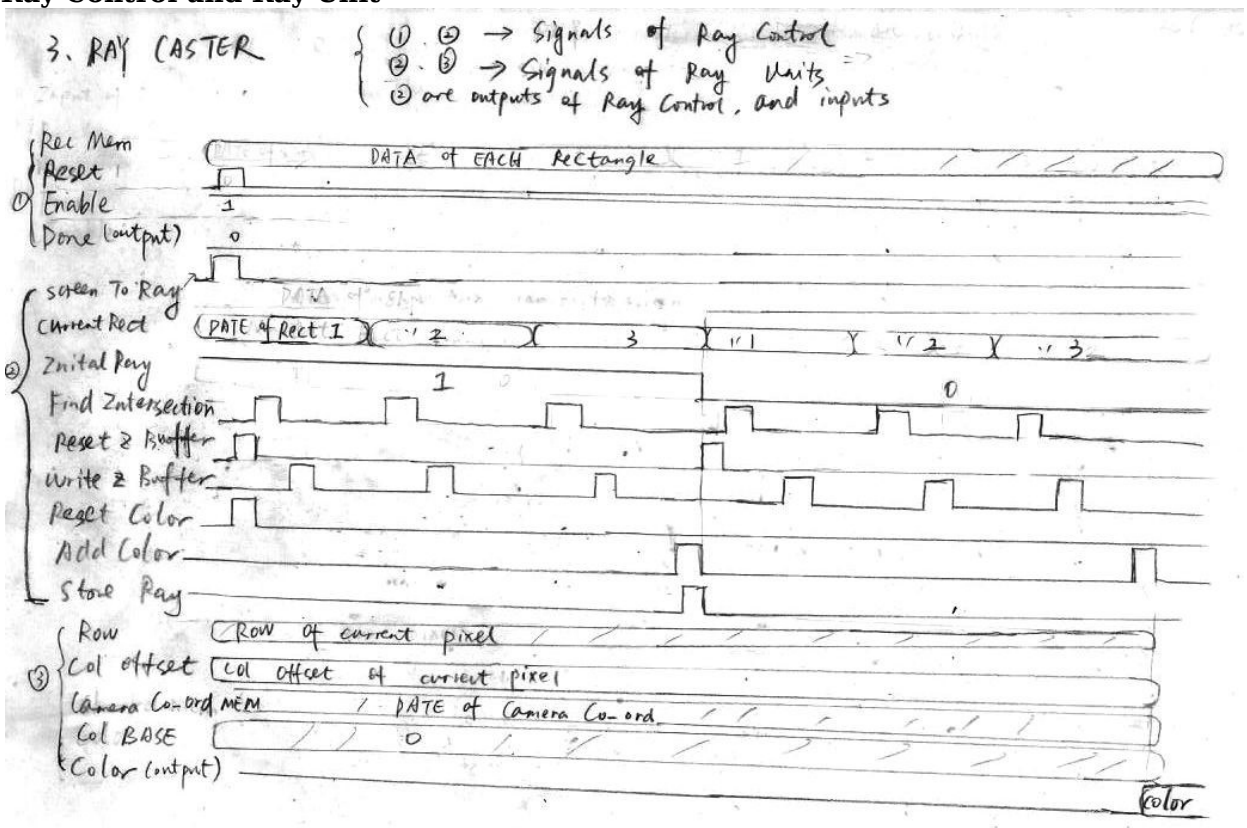
Ray Caster Enable and Reset



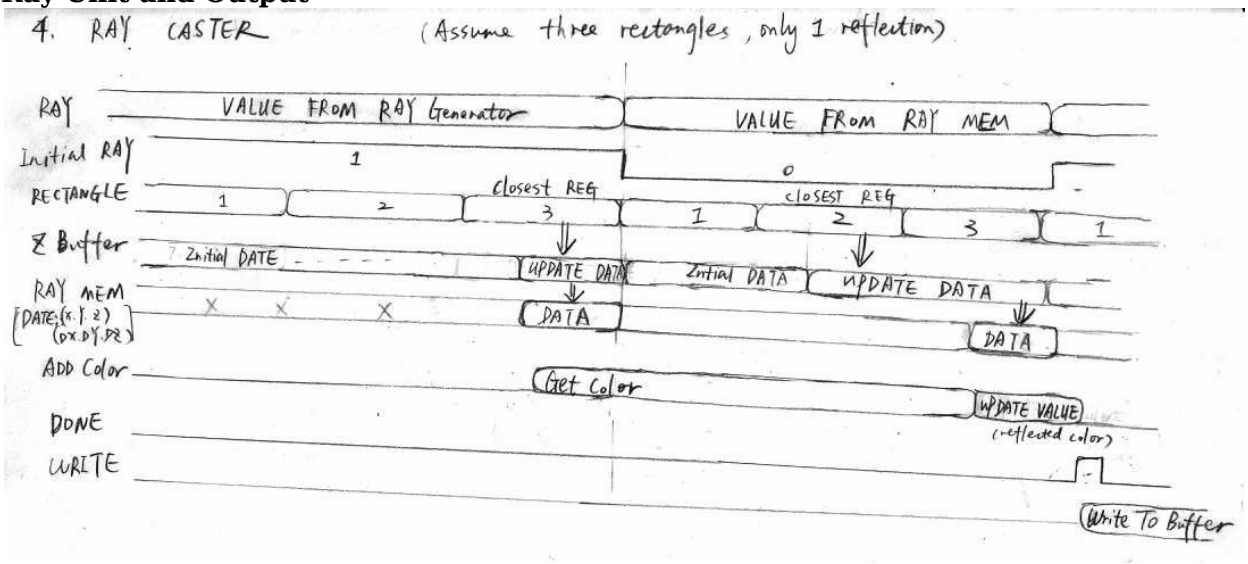
Ray Caster Renderer and Ray Unit



Ray Control and Ray Unit

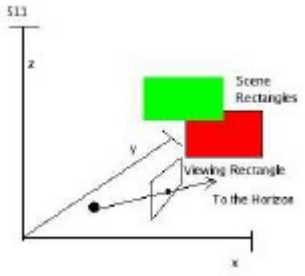


Ray Unit and Output



The Math Behind our Simple, Real-Time Ray Tracer

The Scene



The Coordinate System
The scene is located on an x, y, z coordinate system. Each axis extends from 0 to 511, which just so happens to be 9 bits long, which just so happens to be the width of the on board multipliers on the FPGA.

The Camera
The camera can be placed anywhere. However, the viewing axis of the camera must always be parallel to the x axis. Hence, x is always depth, y is side-to-side, and z is height.

The camera is looking through a viewing rectangle 100 units from the camera along the x-axis. The viewing rectangle is 320 units wide and 240 units tall. The point on the rectangle 50 units from the bottom and 160 units from either side is closest to the camera. Accordingly, the vanishing point of the scene will appear to approach the center of the screen, but towards the bottom.

Scene Rectangles
Our scene consists of a small number of scene rectangles. We describe each scene rectangle with two cordinates which represent 2 oposite corners. Every scene rectangle must be parallel to the plane x = 0, y = 0, or z = 0. For example, the scene rectangle (0, 0, 0) (511, 511, 0) makes a good floor. Each scene rectangle is further described by red, green, and blue color values.

Light
Our entire scene is uniformly lit. Hence, no shadows exist.

Rectangles do no refract. They do, however, reflect off of scene rectangles 50%. Mysteriously, light rays only reflect once or twice.

The Math

For each pixel, we need to determine the color of that pixel. Accordingly, we conceive of a ray which begins at the camera and goes through the corresponding pixel in the viewing rectangle. Then, for each scene rectangle, we determine whether or not our ray intersects that scene rectangle.

Let’s say that our camera is positioned at (cameraX, cameraY, cameraZ). Furthermore we are looking through the point that corresponds to the pixel which is topOffset pixels from

the top of the screen and leftOffset pixels from the left. Accordingly, our ray goes through the point

$$x = cameraX + 100$$
$$y = cameraY - (320/2) + leftOffset$$
$$z = cameraZ + (240 - 50) - topOffset$$

Or in other words, the ray has slope

$$(100, -160 + leftOffset, 190 - topOffset)$$

Recall that the viewing rectangle is always 100 units from the camera along the x-axis. Let us rewrite this slope as

$$(viewX, viewY, viewZ)$$

So our ray can be described by the equations

$$x = cameraX + viewX * t$$
$$y = cameraY + viewY * t$$
$$z = cameraZ + viewZ * t$$

Now consider a scene rectangle which is parallel to the plane y = 0. Its corners are located at (corner1X, cornerY, corner1Z) and (corner2X, cornerY, corner2Z). Notice that the y value for both corners is the same. Our scene rectangle is therefore located on a plane described by the equation

$$y = cornerY$$

Now we must determine where our ray intersects our scene rectangle. Let's call this point (intersectX, intersectY, and intersectZ). Trivially,

$$intersectY = cornerY$$

We simply have to compute intersectX and intersectZ.
Consider our ray where

$$y = cameraY + viewY * t$$

We know y = intersectY = cornerY, so

$$cornerY = cameraY + viewY * t$$

Hence

$$t = (cornerY - cameraY)/viewY$$
$$intersectX = cameraX + viewX * (cornerY - cameraY)/viewY$$
$$intersectZ = cameraZ + viewZ * (cornerY - cameraY)/viewY$$

As you can see, for every ray with every rectangle, we must compute an equation in this form twice:

$$a + b * (c - d)/e$$

We continue by computing the intersection of every scene rectangle with our ray. We determine which is closest by simply picking the smallest

$$|cameraX - intersectX|$$

Reflection

The reflecting ray for a ray and a scene rectangle is simple. Simply determine whether the scene rectangle is parallel to x = 0, y = 0, or z = 0. Then multiply the corresponding portion of the ray by - 1. So in the previous example, our reflecting ray would start at (intersectX, intersectY, intersectZ) and would have slope

$$(100, -1 * (-160 + leftOffset), 190 - topOffset)$$

At this point you can use the same math that we used in the above example to determine color of the reflecting ray.