Simple Hardware Accelerated Real Time Ray Tracer

Dave Smith
Daniel Benamy
Keerti Joshi
Minjie Zhang

Ray Tracer Overview

Ray tracing is a very useful 3D rendering technique. In real life, we see through rays of light that are emitted from a source, colored by reflecting off of objects, and eventually bounced into our eye. The amount and color of light that hits our retina at a certain point thereby allows us to figure out the objects that exist in front of us.

Ray tracing mimics this process, but proceeds in reverse. Rays are shot from our camera, colored when they hit objects, and then bounced toward the light to shade them.

First we generate rays originating from the camera. We follow each ray. For each area the ray passes through we calculate if the ray intersects with an object in that area. If there are no intersections then we continue traversing the next area. If the ray does hit something then we calculate the color of that point. Our ray tracer system does not handle all the functionalities of ray tracing like transparency and refraction.

Our ray tracer system is based on a few assumptions:

1. Our scene is positioned on a (x, y, z) grid.
2. x, y, and z are always greater than 0 and less than 256.
3. Our scene will consist entirely of rectangles.
4. Every rectangle will be parallel to the plane x = 0, y = 0, or z = 0.
5. A diffuse light magically appears from everywhere, so there are no shadows.
6. Light rays mysteriously reflect only once.
7. Every surface contributes 50% of its own color, and 25% reflected color.
8. The camera will always look along an axis parallel to the x-axis. So x is always depth, y is side to side, and z is height.
9. Our scene contains a limited number of rectangles.
10. Rectangles have a single color and no texture.

Design

Hardware Module Hierarchy

- Ray Tracer Project - connects pins
  - Ray Tracer System - built by SOPC Builder
    - NIOS Processor
    - Avalon Bus Stuff
    - SRAM Interface
    - JTAG Debug Interface
    - PS2 Interface - so demo games can use keyboard
    - Ray Tracer Avalon Module - interface between out main module and avalon bus. contains memories.
      - Ray Tracer Renderer - our stuff is in here
      - Rectangle Memory - stores world rectangles
      - Camera Memory - stores camera position

Numeric Representation

In a number of places we need to represent rays in our world space. We do this with 6 signals: X, Y, Z, DX, DY, and DZ. The D variables are the slopes. Coordinates and slopes are represented with 8 bits.

There are also 2 different types of colors we need to represent: rectangle colors and output colors.
Rectangle colors are stored in 1 byte. The breakdown of bits is:
XXRRGGBB
The 'X's are ignored and the R, G, and B values are for red, green, and blue respectively. This is little endian.
Output colors are represented using 18 bits, 6 per color. The disparity between input and output color resolution exists because for a given set of input colors, many more can be produced because of reflections mixing colors.

Timing Constraints

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	20	192,000	0.000005208333333	50MHz	2E-08	**260**

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

We need to render at 60 frames per second because we have to output to the monitor at 60 frames per second and we don't have a full frame buffer that would allow us to render more slowly and reuse frames. We don't have a full frame buffer because it would be very big.

Originally we were hoping to be able to render 12 rectangles in our scene, but that proved to be a little too ambitious. We were able to get in 6 rectangles.

Module Designs

Ray Tracer Renderer

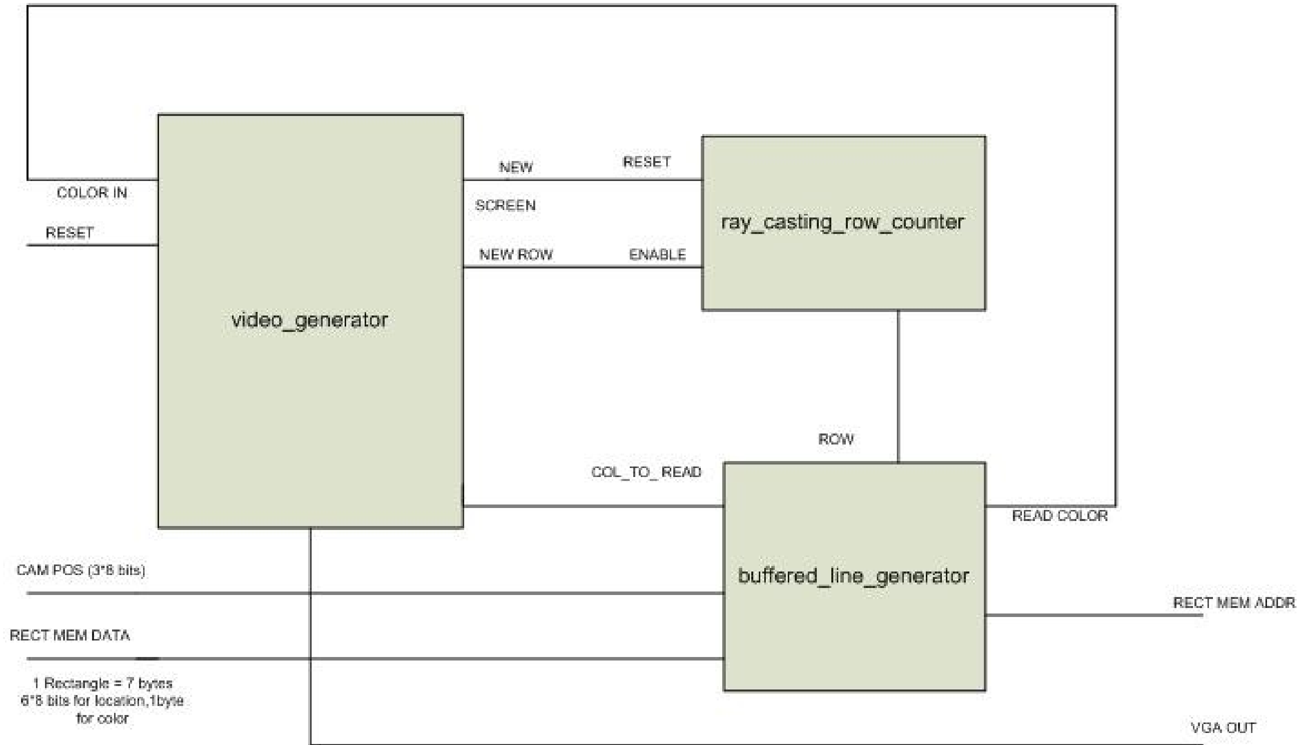


FIG 1

This shows the ray tracer renderer which is composed of the video generator, row counter and the buffered_line_generator. The video generator takes the data produced by the buffered line generator and sends it out to the VGA chip. The video generator is the main driver of the whole system. When it's about to start drawing the screen, it asserts the new screen signal which resets the row counter. Whenever the row counter changes, the buffered line generator wakes up and starts computing the next row of pixels to output.

Video Generator

Our rendering system generates data at a resolution of 320 x 240. Of course it only stores 2 rows at a time, but the row counter and the math is set up for 240 rows and each of those rows is 320 pixels wide. We need to send video out to the monitor at 640 x 480. The video generator takes care of this mismatch. It sends out each pixel of a row twice and goes over each row twice before asserting new row. This allows our data to fill an area 4 times its size.

The vga clock needs to run at around 25 MHz. Our entire system runs at 50 MHz. We decided to even run the video generator internals at 50 MHz to prevent problems when coupling it to the rest of our system. So we have a simple clock divider which only feeds the vga clock output.

Buffered Line Generator

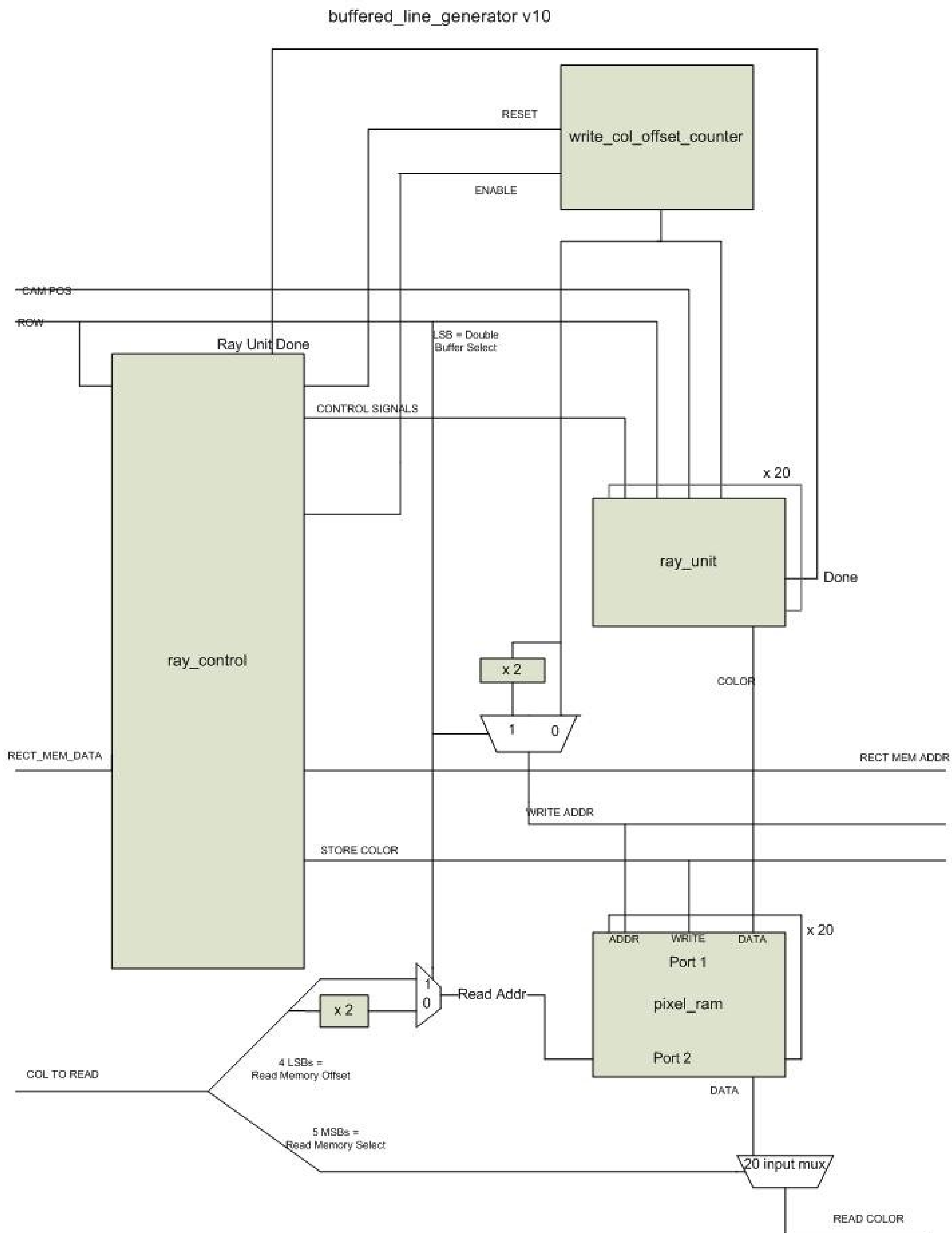


FIG 2

The buffered_line_generator is composed of the ray control, column offset counter, ray units, and pixel memories. The ray control is not a separate vhdl module, but conceptually it is a distinct entity. We

have one block of memory per ray unit which is our double buffer. Having a separate block for each ray unit allows us to write to them in parallel. When we're writing to one area of a memory, the video generator is reading from another. We can do this because they are dual ported memories. So each pixel memory block has room for 2 times the number of pixels a ray unit produces. Which area we read from and write to is controlled using bit zero of the screen row counter so that we alternate each row. We have many ray units which each compute a few columns of output. This parallelism is what allows us to do our rendering in real time.

The diagram only shows a single ray unit and pixel memory for clarity's sake, but in the actual system there are 20 sets in parallel.

When the counter reaches its max value (which is one more than the last value we compute) we'll stop the ray tracer from running again.

The col_to_read line which comes from 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 Control

The ray control coordinates the ray units. One of the jobs of the ray control is to cycle through all the rectangles in the rectangle memory 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.

The Ray Control asserts the correct column offset, camera coordinates, and row. It asserts that there is a new pixel ready to go. Then it asserts the first rectangle. It waits until the ray units are finished, at which point it asserts the next rectangle, and so on, until all the rectangles are finished. Once the rectangles are finished, it asserts that the rectangles are done. It then repeats this whole process for another set of rectangles so the ray units can compute the reflection. Finally, the colors are stored in memory, and the whole process starts again.

Ray Unit

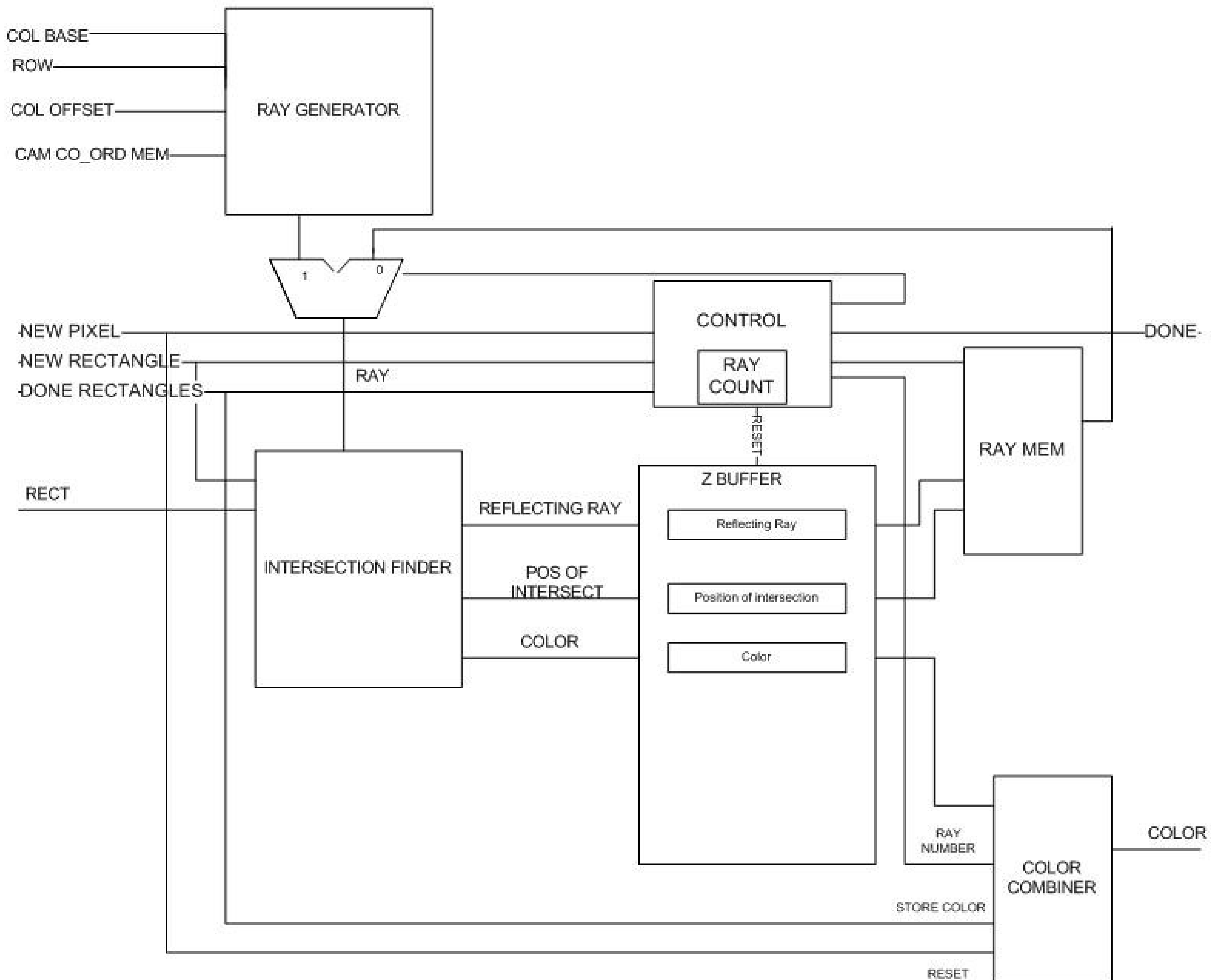


FIG 3

To start off, the new pixel signal will be asserted and ray unit will get cam_co_ord_x, cam_co_ord_y, cam_co_ord_z, x_camera_slope, col_base and row properly. The the new_rectangle signal will be asserted and ray_unit gets the right rectangle coordinates. Some serious math will happen and then we get the calculated X,Y,Z coordinates of the intersection, DX, DY and DZ of the reflected ray, and the color at that intersection. Those values will be compared with previous values to see whether they are closer to the source, and if so, store those new values.

This process will be repeated for every rectangle. After all the rectangles are processed, the done_rectangles signal will be asserted by the Ray Control. This will cause the ColorCombiner within ray_unit grab the resulting color, the closer intersection coordinates and the reflected ray will also be saved.

The entire process will be repeated to handle the reflection except that we will find 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.

Divider

Division was one of the most expensive computations in the ray tracer system. The original version of our divider took 19 clock cycles to finish, one for setup, 18 for dividing. FIG 4, has the block diagram for the basic divider. In each clock cycle of the dividing, we do one subtraction, one right shift of the denominator, and one left shift of the result.

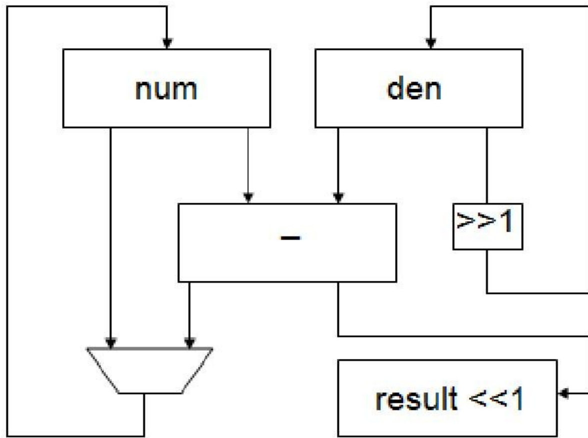
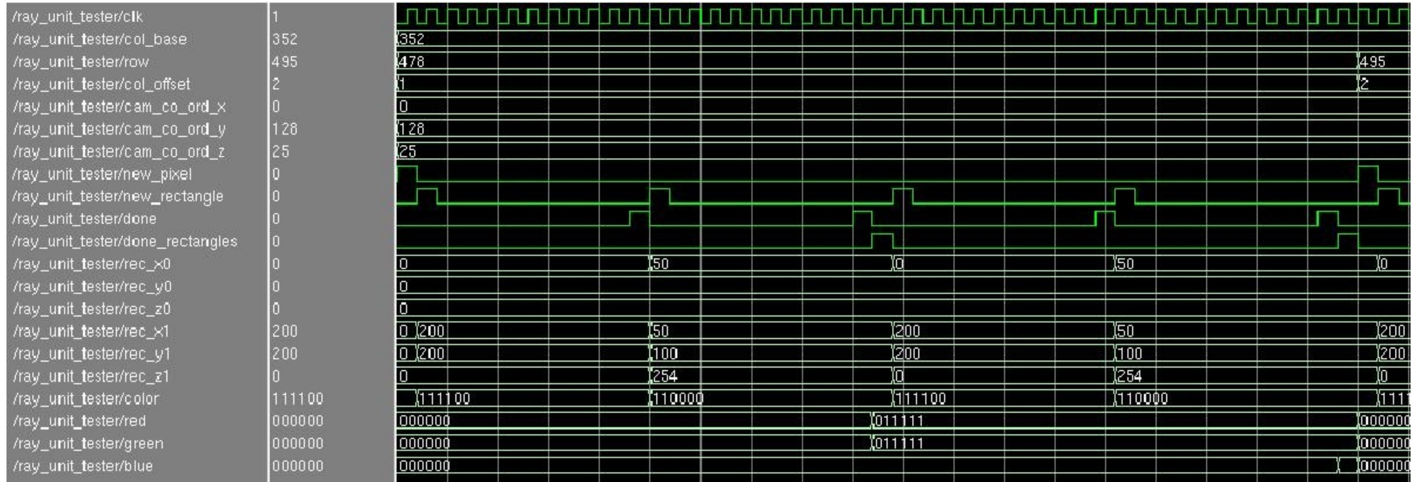


FIG 4 Basic Divider Block

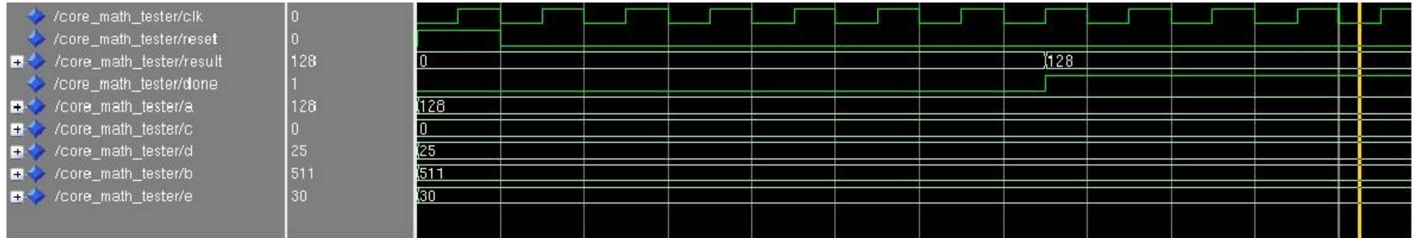
Our current divider optimizes the dividing process by cramming 3 subtractions (as well as shifts of denominator and result) in one cycle, thus we have a 7 cycle divider now, one for setup and only 6 for dividing.

Timing Diagrams

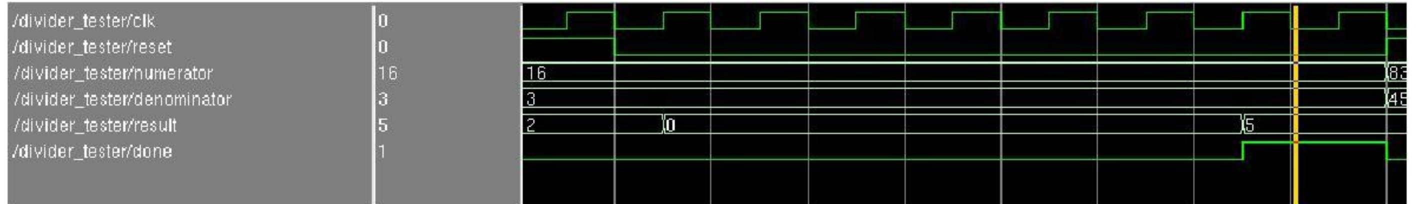
Ray Unit



Core Math



Divider



The Math Behind the System

The Scene

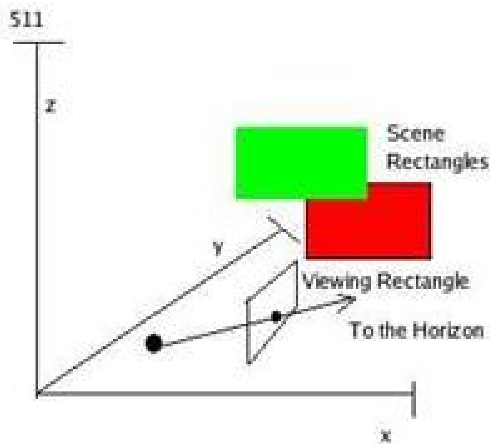


FIG 5 The Coordinate System

The scene is located on an x, y, z coordinate system. Each axis extends from 0 to 256 which allows us to use the 9 bit hardware multipliers with them. Our math actually requires

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 coordinates which represent 2 opposite 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 25%. 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

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

Or in other words, the ray has slope

$$(100, -160 + \text{leftOffset}, 190 - \text{topOffset})$$

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

$$(\text{viewX}, \text{viewY}, \text{viewZ})$$

So our ray can be described by the equations

$$x = \text{cameraX} + \text{viewX} * t$$

$$y = \text{cameraY} + \text{viewY} * t$$

$$z = \text{cameraZ} + \text{viewZ} * t$$

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

$$y = \text{cornerY}$$

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

$$\text{intersectY} = \text{cornerY}$$

We simply have to compute intersectX and intersectZ.

Consider our ray where

$$y = \text{cameraY} + \text{viewY} * t$$

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.

Testing and Debugging

Testing and debugging was one of the important aspects in building our system. For each module in the system we have test fixtures which test the module extensively with a large number of stimulus and emulates the hardware as closely as possible. The test fixtures set up specific input and then use assertions to make sure the outputs are correct. This made running regression tests and debugging easier.

For many of the components, we instrumented the C prototype to generate a scene and print out tons and tons of lines of vhdl tests which verify that our hardware has the same results as the prototype. The C prototype (without all the printing code as we didn't keep it all) and an example of the generated tests are included in the file listing.

Software

We wrote some simple games and animations in software to demonstrate the capabilities of the hardware (and to impress our friends). They are written in C and run on the nios processor. The software module controls the position and movements of the rectangles on the screen by writing to memory mapped registers of our hardware.

Project Management

Version Control

We used revision control for all project code and documentation. Most of it is in a subversion repository hosted by assembla which is a free non-open source project hosting site. We've also got a couple of documents in google docs. We played around with assembla's bug tracking system, but didn't use it too much in the end. The repository can be accessed online at http://svn2.assembla.com/svn/hwraytracer/.

Team Work

Non-coding Work
Initial architecture: Dan. Subsequent revisions: all.
Algorithm design: Dave
Documentation: Keerti, Minjie, Dan

Implementation
C Prototype: Dave
Divider: Keerti, Minjie, Dave
Core math: Dave
Ray unit: Dave
Ray control: Dave, Minjie
Ray tracer renderer: Dan
Video generator: Keerti, Dan
Avalon stuff: Dan
Software: Dan, Dave

A Piece of Advice

Dave: The coffee in the Uris cafe is better than in Mudd.

Dan: Use revision control. Use assertions to make testing easier. Set aggressive deadlines; everything takes longer than expected and if you schedule some slack at the end, you can polish the project if you're on schedule and have time to spend debugging if you're behind.

Keerti: Start early and have specific times set to work on the project so that there is constant progress and deadlines are easily met. Discuss your ideas about the project implementation with prof. Edwards since he can guide you well and give a lot of clever ideas as well. Have well defined test cases and test benches for all modules in the project which makes debugging a lot easier. Do not ignore version control.Use subversion so that the latest versions of the code and documentation are easily accessible by all team members.

Minjie: To make the code function correctly is far from perfect. We should make it easy to read, easy to edit, and optimize it in order to save memory and reduce computations required. Such as the divider and core_math unit in our project, we get a 18 cycle divider at first, and then make it 9 cycle, and 6 cycle at last. Also, by optimizing the core_math unit, we need less computations and thus be able to allow more rectangles working together.

File Listings

ray_tracer_project.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ray_tracer_project is
	port (
		signal CLOCK_50 : IN STD_LOGIC;
		--signal reset_n : IN STD_LOGIC;
		signal SRAM_ADDR : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);
		signal SRAM_DQ :  INOUT STD_LOGIC_VECTOR (15 DOWNTO 0);
		signal SRAM_CE_N : OUT STD_LOGIC;
		signal SRAM_LB_N : OUT STD_LOGIC;
		signal SRAM_OE_N : OUT STD_LOGIC;
		signal SRAM_UB_N : OUT STD_LOGIC;
		signal SRAM_WE_N : OUT STD_LOGIC;
		signal VGA_CLK : OUT STD_LOGIC;
		signal VGA_HS : OUT STD_LOGIC;
		signal VGA_VS : OUT STD_LOGIC;
		signal VGA_BLANK : OUT STD_LOGIC;
		signal VGA_SYNC : OUT STD_LOGIC;
		signal VGA_R : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
		signal VGA_G : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
		signal VGA_B : OUT STD_LOGIC_VECTOR (9 DOWNTO 0);
		signal SW : IN STD_LOGIC_VECTOR (17 DOWNTO 0);
		signal LEDR : OUT STD_LOGIC_VECTOR (17 DOWNTO 0);

		signal PS2_CLK : INOUT STD_LOGIC;
		signal PS2_DAT : INOUT STD_LOGIC
	);
end entity ray_tracer_project;
architecture rtl of ray_tracer_project is
	signal rect_memory_address : unsigned (7 downto 0) := (others => '0');
	signal screen_col_color : std_logic_vector (17 downto 0);
	signal done : std_logic := '0';
begin

	core: entity work.ray_tracer_system port map (
		clk => CLOCK_50,
		reset_n => '1', --KEY0,
		SRAM_ADDR_from_the_de2_sram_controller_inst => SRAM_ADDR,
		SRAM_CE_N_from_the_de2_sram_controller_inst => SRAM_CE_N,
		SRAM_DQ_to_and_from_the_de2_sram_controller_inst => SRAM_DQ,
		SRAM_LB_N_from_the_de2_sram_controller_inst => SRAM_LB_N,
		SRAM_OE_N_from_the_de2_sram_controller_inst => SRAM_OE_N,
		SRAM_UB_N_from_the_de2_sram_controller_inst => SRAM_UB_N,
		SRAM_WE_N_from_the_de2_sram_controller_inst => SRAM_WE_N,
		vga_b_from_the_ray_tracer_avalon_module_inst => VGA_B,
		vga_blank_from_the_ray_tracer_avalon_module_inst => VGA_BLANK,
		vga_clock_from_the_ray_tracer_avalon_module_inst => VGA_CLK,
		vga_g_from_the_ray_tracer_avalon_module_inst => VGA_G,
		vga_h_sync_from_the_ray_tracer_avalon_module_inst => VGA_HS,
		vga_r_from_the_ray_tracer_avalon_module_inst => VGA_R,
		vga_sync_from_the_ray_tracer_avalon_module_inst => VGA_SYNC,
		vga_v_sync_from_the_ray_tracer_avalon_module_inst => VGA_VS,
		debug_address_to_the_ray_tracer_avalon_module_inst => SW(6 downto 0),
		debug_data_from_the_ray_tracer_avalon_module_inst => LEDR(8 downto 0),
		PS2_CLK_to_and_from_the_Altera_UP_Avalon_PS2_inst => PS2_CLK,
		PS2_DAT_to_and_from_the_Altera_UP_Avalon_PS2_inst => PS2_DAT
```

```
        );
end rtl;
```

---

ray_tracer_avalon_module.vhd

```
--------------------------------------------------------------------------------
-- Connects to Avalon system bus from NIOS processor.
-- Memory interface:
-- Address  Bits  Direction  Description  Valid values
-- 0        16    R/W        Camera X     0 - 511
-- 1        16    R/W        Camera Y     0 - 511
-- 2        16    R/W        Camera Z     0 - 511
--
-- The following values can fit in 8 bits but the interface is still 16 bits
-- wide and they should be written as 16 bit values. This is so all the
-- registers are the same size and to make the transition easier if we want to
-- expand the range later.
-- 3        16    R/W        Rect 1 X1    0 - 255
-- 4        16    R/W        Rect 1 Y1    0 - 255
-- 5        16    R/W        Rect 1 Z1    0 - 255
-- 6        16    R/W        Rect 1 X2    0 - 255
-- 7        16    R/W        Rect 1 Y2    0 - 255
-- 8        16    R/W        Rect 1 Z2    0 - 255
-- 9        16    R/W        Rect 1 Color 0 - 2?
--
-- 10                        Rect 2
--
-- 17                        Rect 3
-- We needed to split the entity and architecture into 2 files because SOPC
-- builder couldn't use the file with the architecture in it.
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ray_tracer_avalon_module is
        port (
                reset : in std_logic := '0';
                clock : in std_logic := '0';

                -- Avalon Interface
                read       : in  std_logic := '0';
                write      : in  std_logic := '0';
                chipselect : in  std_logic := '0';
                address    : in  unsigned(6 downto 0)  := (others => '0');
                readdata   : out unsigned(15 downto 0) := (others => '0');
                writedata  : in  unsigned(15 downto 0) := (others => '0');
                -- VGA Outputs
                vga_clock,
                vga_h_sync,
                vga_v_sync,
                vga_blank,
                vga_sync : out std_logic := '0';
                vga_r,
                vga_g,
                vga_b : out unsigned(9 downto 0) := (others => '0');
                -- Debugging - access to camera and rectangle memories
                debug_address : in unsigned(6 downto 0) := (others => '0');
                debug_data : out unsigned(8 downto 0) := (others => '0')
        );
```

```
end ray_tracer_avalon_module;
```

---

ray_tracer_avalon_module_arch.vhd

```
-- See documentation in ray_tracer_avalon_module.vhd
architecture rtl of ray_tracer_avalon_module is

        signal camera_x : unsigned (8 downto 0);
        signal camera_y : unsigned (8 downto 0);
        signal camera_z : unsigned (8 downto 0);
        signal camera_x_buffer : unsigned (8 downto 0);
        signal camera_y_buffer : unsigned (8 downto 0);
        signal camera_z_buffer : unsigned (8 downto 0);
        type rectangle_memory_type is array (84 downto 0) of unsigned (8 downto 0);
 -- we really only need 13, but then simulation fails because we try to read one past
the end.
        signal rectangle_memory : rectangle_memory_type := (others => "000000000");
        signal rectangle_memory_buffer : rectangle_memory_type := (others =>
"000000000");
        signal buffer_index : unsigned (6 downto 0) := (others => '0');
        -- See ray_control for description:
        signal rect_memory_address : unsigned (7 downto 0);
        signal rect_memory_data : unsigned (8 downto 0);

        signal start_screen : std_logic := '0';
begin
        ray_tracer_renderer_instance : entity work.ray_tracer_renderer port map (
                clock => clock,
                reset => reset,
                camera_x => camera_x,
                camera_y => camera_y,
                camera_z => camera_z,
                rect_memory_address => rect_memory_address,
                rect_memory_data => rect_memory_data,
                vga_clock => vga_clock,
                vga_h_sync => vga_h_sync,
                vga_v_sync => vga_v_sync,
                vga_blank => vga_blank,
                vga_sync => vga_sync,
                vga_r => vga_r,
                vga_g => vga_g,
                vga_b => vga_b,

                start_screen_out => start_screen
        );
        BusInterface : process (clock, reset)
        begin
                if reset = '1' then
                        camera_x <= "000000000";
                        camera_y <= "000000000";
                        camera_z <= "000000000";
                elsif rising_edge(clock) then
                        if chipselect = '1' and write = '1' then
                                if address = 0 then
                                        camera_x_buffer <= writedata(8 downto 0);
                                elsif address = 1 then
                                        camera_y_buffer <= writedata(8 downto 0);
                                elsif address = 2 then
                                        camera_z_buffer <= writedata(8 downto 0);
                                else
```

```
                                        rectangle_memory_buffer(to_integer(address -
3)) <= writedata(8 downto 0);
                                end if;
                        elsif chipselect = '1' and read = '1' then
                                if address = 0 then
                                        readdata <= "0000000" & camera_x_buffer;
                                elsif address = 1 then
                                        readdata <= "0000000" & camera_y_buffer;
                                elsif address = 2 then
                                        readdata <= "0000000" & camera_z_buffer;
                                else
                                        readdata <= "0000000" &
rectangle_memory_buffer(to_integer(address - 3));
                                end if;
                        end if;
                        if start_screen = '1' then
                                camera_x <= camera_x_buffer;
                                camera_y <= camera_y_buffer;
                                camera_z <= camera_z_buffer;
                                buffer_index <= "0000000";
                        elsif buffer_index /= "1010101" then -- 85
                                rectangle_memory(to_integer(buffer_index)) <=
rectangle_memory_buffer(to_integer(buffer_index));
                                buffer_index <= buffer_index + 1;
                        end if;
                end if;
        end process BusInterface;

        rectangle_memory_access : process (clock)
        begin
                if rising_edge(clock) then
                        rect_memory_data <=
rectangle_memory(to_integer(rect_memory_address));
                end if;
        end process rectangle_memory_access;

        debugging : process (clock)
        begin
                if rising_edge(clock) then
                        if debug_address = 0 then
                                debug_data <= camera_x;
                        elsif debug_address = 1 then
                                debug_data <= camera_y;
                        elsif debug_address = 2 then
                                debug_data <= camera_z;
                        else
                                debug_data <=
rectangle_memory(to_integer(debug_address - 3));
                        end if;
                end if;
        end process debugging;
end rtl;
```

---

ray_tracer_avalon_module_tester.vhd

```
-- It's gonna be pretty goddamn hard to automatically verify that the outputs
-- are right. Instead, we'll just throw some reads and writes at it and take a
-- look at the wave forms to make sure the right things are happening.
library ieee;
use ieee.std_logic_1164.all;
```

```
use ieee.numeric_std.all;
entity ray_tracer_avalon_module_tester is
end ray_tracer_avalon_module_tester;
architecture test of ray_tracer_avalon_module_tester is
        signal clock : std_logic := '0';
        signal reset : std_logic := '1';
        signal read       : std_logic := '0';
        signal write      : std_logic := '0';
        signal chipselect : std_logic := '0';
        signal address    : unsigned(6 downto 0)  := (others => '0');
        signal readdata   : unsigned(15 downto 0) := (others => '0');
        signal writedata  : unsigned(15 downto 0) := (others => '0');
        signal vga_clock : std_logic  := '0';
        signal vga_h_sync : std_logic := '0';
        signal vga_v_sync : std_logic := '0';
        signal vga_blank : std_logic  := '0';
        signal vga_sync : std_logic   := '0';
        signal vga_r : unsigned(9 downto 0) := (others => '0');
        signal vga_g : unsigned(9 downto 0) := (others => '0');
        signal vga_b : unsigned(9 downto 0) := (others => '0');
begin
        ray_tracer_avalon_module_inst : entity work.ray_tracer_avalon_module port map
(
                clock => clock,
                reset => reset,
                read => read,
                write => write,
                chipselect => chipselect,
                address => address,
                readdata => readdata,
                writedata => writedata,
                vga_clock => vga_clock,
                vga_h_sync => vga_h_sync,
                vga_v_sync => vga_v_sync,
                vga_blank => vga_blank,
                vga_sync => vga_sync,
                vga_r => vga_r,
                vga_g => vga_g,
                vga_b => vga_b,

                debug_address => (others => '0')
        );
        process
        begin
                loop
                        clock <= '0';
                        wait for 10 ns;
                        clock <= '1';
                        wait for 10 ns;
                        reset <= '0';
                end loop;
        end process;

        process
        begin
                wait for 20 ns;
                read <= '0';
                chipselect <= '1';

                -- Camera
                address <= "0000000"; writedata <= "0000000011100111"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0000001"; writedata <= "0000000010101011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
```

```
                address <= "0000010"; writedata <= "0000000000001010"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;

                -- Rectangles
                address <= "0000011"; writedata <= "0000000000000000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0000100"; writedata <= "0000000000000000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0000101"; writedata <= "0000000000000000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0000110"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0000111"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001000"; writedata <= "0000000000000000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001001"; writedata <= "0000000000111100"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001010"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001011"; writedata <= "0000000000000000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001100"; writedata <= "0000000000000000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001101"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001110"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0001111"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010000"; writedata <= "0000000000000011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010001"; writedata <= "0000000011110001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010010"; writedata <= "0000000010100001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010011"; writedata <= "0000000000000001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010100"; writedata <= "0000000011111011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010101"; writedata <= "0000000010110101"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010110"; writedata <= "0000000000000001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0010111"; writedata <= "0000000000001100"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011000"; writedata <= "0000000011110001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011001"; writedata <= "0000000010100001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011010"; writedata <= "0000000000001011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011011"; writedata <= "0000000011111011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011100"; writedata <= "0000000010110101"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011101"; writedata <= "0000000000001011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011110"; writedata <= "0000000000001100"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0011111"; writedata <= "0000000011110001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100000"; writedata <= "0000000010100001"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
```

```
                address <= "0100001"; writedata <= "0000000000010101"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100010"; writedata <= "0000000011111011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100011"; writedata <= "0000000010110101"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100100"; writedata <= "0000000000010101"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100101"; writedata <= "0000000000001100"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100110"; writedata <= "0000000011001000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0100111"; writedata <= "0000000000110010"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0101000"; writedata <= "0000000011111110"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0101001"; writedata <= "0000000011001000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0101010"; writedata <= "0000000011001000"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0101011"; writedata <= "0000000011111011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;
                address <= "0101100"; writedata <= "0000000000110011"; write <= '1';
wait for 20 ns; write <= '0'; wait for 20 ns;

                wait for 20 ms;

                report "No problem mon! End of simulation." severity failure;
        end process;
end test;
```

---

ray_tracer_renderer.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ray_tracer_renderer is
port(
        clock : in std_logic;
        reset : in std_logic;
        camera_x,
        camera_y,
        camera_z : in unsigned (8 downto 0);

        -- Allow this module to request rect data from whatever contains it. See
        -- ray_control for more info.
        rect_memory_address : out unsigned (7 downto 0);
        rect_memory_data : in unsigned (8 downto 0);

        vga_clock,
        vga_h_sync,
        vga_v_sync,
        vga_blank,
        vga_sync : out std_logic;
        vga_r,
        vga_g,
        vga_b : out unsigned(9 downto 0);

        start_screen_out : out std_logic := '0'
);
```

```
end ray_tracer_renderer ;
architecture tracer of ray_tracer_renderer is
        constant INITIAL_ROW : signed(8 downto 0) := "010111101"; -- 189
        constant LAST_ROW : signed(8 downto 0) := "111001110"; -- - 50

        signal screen_column : unsigned (8 downto 0);
-- column video generator wants to read
        signal screen_color : std_logic_vector (17 downto 0);
-- color of pixel at said column
        signal start_screen : std_logic;
        signal start_row : std_logic;
        signal last_start_row : std_logic;
        signal ray_casting_row_counter : signed (8 downto 0);
        signal row_minus_1 : signed (8 downto 0);
begin
        start_screen_out <= start_screen;

        video_generator : entity work.video_generator port map (
                clock => clock,
                reset => reset,

                col_to_read => screen_column,
                color => screen_color,
                start_screen => start_screen,
                start_row => start_row,
                vga_clock => vga_clock,
                vga_h_sync => vga_h_sync,
                vga_v_sync => vga_v_sync,
                vga_blank => vga_blank,
                vga_sync => vga_sync,
                vga_r => vga_r,
                vga_g => vga_g,
                vga_b => vga_b
        );

        row_counter : process(clock)
                variable row : signed (8 downto 0) := (others => '0');
        begin
                if rising_edge(clock) then
                        row := ray_casting_row_counter;
                        if start_screen = '1' or reset = '1' then -- reset
                                row := INITIAL_ROW;
                        elsif start_row = '1' and last_start_row = '0' then -- enable
                        -- start_row is based on the 25 mhz clock, so over here it
looks
                        -- like it's high for 2 cycles. last_start_row prevents us
from
                        -- incrementing the counter twice.
                                row := row - 1;
                                last_start_row <= '1';
                        else
                                last_start_row <= '0';
                        end if;
                        ray_casting_row_counter <= row;
                        if row = LAST_ROW then
                                row_minus_1 <= INITIAL_ROW;
                        else
                                row_minus_1 <= row - 1;
                        end if;
                end if;
        end process;

        buffered_line_generator : entity work.buffered_line_generator port map (
                clock => clock,
```

```
                row => row_minus_1,

                camera_x => camera_x,
                camera_y => camera_y,
                camera_z => camera_z,

                rect_memory_address => rect_memory_address,
                rect_memory_data => rect_memory_data(7 downto 0),

                screen_col => screen_column,
                screen_col_color => screen_color
        );
end tracer;
```

----

ray_unit.vhd

```
library ieee;

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ray_unit is
        port (
                signal clk : in std_logic;

                -- initial ray
                signal col_base : in signed (8 downto 0) := (others => '0');
-- should be one of these: -160, -150, ... , -10, 0, 10, ... , 150
                signal row : in signed (8 downto 0) := (others => '0');
-- should start at 189 and go down to -50
                signal col_offset : in unsigned (3 downto 0) := (others => '0');
-- should start at 0 and go to 9
                signal cam_co_ord_x, cam_co_ord_y, cam_co_ord_z :
        in unsigned (8 downto 0) := (others => '0');
 -- coordinates of the camera

                -- control signals
                signal new_pixel : in std_logic := '0';
-- assert for 1 cycle when starting a new pixel
                signal new_rectangle : in std_logic := '0';
-- assert for 1 cycle with each new rectangle
                signal done : out std_logic := '0';
-- active high. Wait for this signal after you assert new_rectangle
                signal done_rectangles : in std_logic := '0';
-- assert for 1 cycle when you finish all the rectangles

                -- rectangles are 7 bytes. x,y,z for corners 1,2; and 1 byte for
color.
                signal rec_x0, rec_y0, rec_z0, rec_x1, rec_y1, rec_z1 :
       in unsigned (8 downto 0) := (others => '0');
                -- color(2) = red, color(1) = green, color(0) = blue. Or, "rgb"
                signal color : in unsigned (5 downto 0) := (others => '0');

                -- color output
                -- this is the final result.
                -- tests are written for separate red, green, blue. Don't actually
use.
                signal red, green, blue : out unsigned (5 downto 0) := (others =>
'0');
                signal rgb : out unsigned (17 downto 0) := (others => '0')
```

```
        );
end ray_unit;
architecture unit of ray_unit is
        -- current ray.
        constant X_CAMERA_SLOPE : signed(8 downto 0) := "000110010";
        signal ray_x, ray_y, ray_z : unsigned (8 downto 0) := (others => '0');
        signal ray_dx, ray_dy, ray_dz : signed (8 downto 0) := (others => '0');
        -- core math
        signal a1, c1, d1 : unsigned (8 downto 0) := (others => '0');
        signal b1, e1 : signed (8 downto 0) := (others => '0');
        signal core_result1 : unsigned (8 downto 0) := (others => '0');
        signal core_1_done : std_logic := '0';
        signal a2, c2, d2 : unsigned (8 downto 0) := (others => '0');
        signal b2, e2 : signed (8 downto 0) := (others => '0');
        signal core_result2 : unsigned (8 downto 0) := (others => '0');
        signal core_2_done : std_logic := '0';
        signal reset_core : std_logic := '0';
        -- intersection_finder
        type states is (zero, one, two, three);
        signal state : states;
        signal intersect_x, intersect_y, intersect_z :
        unsigned (8 downto 0) := (others => '0');
        -- z_buffer
        signal close_distance : unsigned (8 downto 0) := (others => '0');
        signal close_x, close_y, close_z : unsigned (8 downto 0) := (others => '0');
        signal close_r, close_g, close_b : unsigned (8 downto 0) := (others => '0');
        signal reflect_x, reflect_y, reflect_z : signed (8 downto 0) := (others =>
'0');
        -- color_combiner
        signal ray_count : unsigned (0 downto 0) := "0";
        signal total_r, total_g, total_b : unsigned (8 downto 0) := (others => '0');

        constant INFINITE : unsigned (8 downto 0) := "011111111";
begin
        red <= total_r(7 downto 2);
        green <= total_g(7 downto 2);
        blue <= total_b(7 downto 2);
        rgb <= total_r(7 downto 2) & total_g(7 downto 2) & total_b(7 downto 2);

        depending_on_control : process(clk)
                variable blue_helper : signed (8 downto 0) := (others => '0');
                variable temp_r_0 : unsigned (1 downto 0) := (others => '0');
                variable temp_r_1 : unsigned (6 downto 0) := (others => '0');
                variable temp_g_0 : unsigned (1 downto 0) := (others => '0');
                variable temp_g_1 : unsigned (6 downto 0) := (others => '0');
                variable temp_b_0 : unsigned (1 downto 0) := (others => '0');
                variable temp_b_1 : unsigned (6 downto 0) := (others => '0');
        begin
                if rising_edge(clk) then
                        if new_pixel = '1' then
                                ray_x <= cam_co_ord_x;
                                ray_y <= cam_co_ord_y;
                                ray_z <= cam_co_ord_z;

                                ray_dx <= X_CAMERA_SLOPE;
                                ray_dy <= col_base + signed("0" & col_offset);
                                ray_dz <= row;
                                reflect_x <= X_CAMERA_SLOPE;
                                reflect_y <= col_base + signed("0" & col_offset);
                                reflect_z <= row;

                                total_r <= "000000000";
                                total_g <= "000000000";
                                total_b <= "000000000";
```

```
						ray_count <= "0";
				elsif done_rectangles = '1' then
						ray_x <= close_x;
						ray_y <= close_y;
						ray_z <= close_z;
						ray_dx <= reflect_x;
						ray_dy <= reflect_y;
						ray_dz <= reflect_z;

						if ray_count = "0" then
								total_r <= close_r / 2;
								total_g <= close_g / 2;
								total_b <= close_b / 2;
						elsif ray_count = "1" then
								total_r <= total_r + close_r / 4;
								total_g <= total_g + close_g / 4;
								total_b <= total_b + close_b / 4;
						end if;
						ray_count <= ray_count + 1;
				end if;

				-- Handle next rectangle
				if new_rectangle = '1' then
						if rec_x0 = rec_x1 then
								intersect_x <= rec_x0;
								a1 <= ray_y; b1 <= ray_dy; c1 <= rec_x0;
d1 <= ray_x; e1 <= ray_dx;a2 <= ray_z;
b2 <= ray_dz; c2 <= rec_x0; d2 <= ray_x;
 e2 <= ray_dx;
						elsif rec_y0 = rec_y1 then
								a1 <= ray_x; b1 <= ray_dx; c1 <= rec_y0;
 d1 <= ray_y; e1 <= ray_dy;
 intersect_y <= rec_y0; a2 <= ray_z;
b2 <= ray_dz; c2 <= rec_y0;
d2 <= ray_y; e2 <= ray_dy;
elsif rec_z0 = rec_z1 then
								a1 <= ray_x; b1 <= ray_dx; c1 <= rec_z0;
d1 <= ray_z; e1 <= ray_dz; a2 <= ray_y;
b2 <= ray_dy; c2 <= rec_z0; d2 <= ray_z;
 e2 <= ray_dz; intersect_z <= rec_z0;
						end if;
						reset_core <= '1';
						state <= zero;
						done <= '0';
				else
						case state is
						when zero =>
								reset_core <= '0';
								state <= one;
						when one =>
								if core_1_done = '1' and core_2_done = '1'
then
										if rec_x0 = rec_x1 then
												intersect_y <= core_result1;
												intersect_z <= core_result2;
										elsif rec_y0 = rec_y1 then
												intersect_x <= core_result1;
												intersect_z <= core_result2;
										elsif rec_z0 = rec_z1 then
												intersect_x <= core_result1;
												intersect_y <= core_result2;
										end if;
										state <= two;
								end if;
```

```
when two =>
        if intersect_x /= INFINITE
        and intersect_y /= INFINITE
        and intersect_z /= INFINITE
        and intersect_x /= ray_x
        and (
                (intersect_x <= rec_x0
                  and intersect_x >= rec_x1)
                  or intersect_x >= rec_x0)
                 and intersect_x <= rec_x1)
        )
        and (
                (intersect_y <= rec_y0
                   and intersect_y >= rec_y1)
                 or (intersect_y >= rec_y0
                and intersect_y <= rec_y1)
        ) and (
                (intersect_z <= rec_z0
                and intersect_z >= rec_z1) or
                (intersect_z >= rec_z0
                and intersect_z <= rec_z1)
        ) and (
                (intersect_x < ray_x
                and ray_x - intersect_x
                < close_distance) or
                (intersect_x > ray_x
                and intersect_x - ray_x
                < close_distance)
        ) then
        if (intersect_x < ray_x) then
                close_distance <= ray_x -
intersect_x;
        else
                close_distance <= intersect_x
- ray_x;
        end if;
        close_x <= intersect_x;
        close_y <= intersect_y;
        close_z <= intersect_z;
        close_r <= "000000000";
        close_g <= "000000000";
        close_b <= "000000000";

        temp_r_0 := "00";
        if color(5) = '1' then
                temp_r_0 := "01";
        end if;
        temp_r_1 := "0000000";
        if color(4) = '1' then
                temp_r_1 := "1111111";
        end if;
        close_r <= temp_r_0 & temp_r_1;
        temp_g_0 := "00";
        if color(3) = '1' then
                temp_g_0 := "01";
        end if;
        temp_g_1 := "0000000";
        if color(2) = '1' then
                temp_g_1 := "1111111";
        end if;
        close_g <= temp_g_0 & temp_g_1;
        temp_b_0 := "00";
        if color(1) = '1' then
                temp_b_0 := "01";
```

```
                                        end if;
                                        temp_b_1 := "0000000";
                                        if color(0) = '1' then
                                                temp_b_1 := "1111111";
                                        end if;
                                        close_b <= temp_b_0 & temp_b_1;

                                        if rec_x0 = rec_x1 then
                                                reflect_x <= not(ray_dx) + 1;
                                                reflect_y <= ray_dy;
                                                reflect_z <= ray_dz;
                                        elsif rec_y0 = rec_y1 then
                                                reflect_x <= ray_dx;
                                                reflect_y <= not(ray_dy) + 1;
                                                reflect_z <= ray_dz;
                                        elsif rec_z0 = rec_z1 then
                                                reflect_x <= ray_dx;
                                                reflect_y <= ray_dy;
                                                reflect_z <= not(ray_dz) + 1;
                                        end if;
                                end if;
                                done <= '1';
                                state <= three;
                        when three =>
                                -- hang out for a while
                        end case;
                end if;

                -- Reset closest rectangle
                if new_pixel = '1' or done_rectangles = '1' then
                        done <= '0';
                        if new_pixel = '1' then
                                blue_helper := row;
                        elsif done_rectangles = '1' then
                                blue_helper := reflect_z;
                        end if;
                        blue_helper := blue_helper + "000110010";
                        -- reflect_z can be less than -50. In that case, the
background color
                        -- should be black, just like if we were at row -50.
                        if blue_helper(8) = '1' and done_rectangles = '1'
 and reflect_z(8) = '1' then
                                blue_helper := "000000000";
                        end if;
                        close_distance <= INFINITE;
                        close_x <= INFINITE; close_y <= INFINITE; close_z <=
INFINITE;
                        close_r <= "000000000";
                        close_g <= "000000000";
                        close_b <= unsigned(blue_helper);
                end if;
        end if;
end process;

intersection_1 : entity work.core_math
port map (
        clk => clk,
        reset => reset_core,
        a => a1,
        b => b1,
        c => c1,
        d => d1,
        e => e1,
        result => core_result1,
```

```
                done => core_1_done
        );

        intersection_2 : entity work.core_math
        port map (
                clk => clk,
                reset => reset_core,
                a => a2,
                b => b2,
                c => c2,
                d => d2,
                e => e2,
                result => core_result2,
                done => core_2_done
        );
end unit;

--------------------------------------------------------------------------------------
```

buffered_line_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity buffered_line_generator is
        port(
                signal clock : in std_logic := '0';

                signal row : in signed (8 downto 0) := (others => '0');
                signal camera_x, camera_y, camera_z :
       in unsigned (8 downto 0) := (others => '0');

                signal rect_memory_address :
        out unsigned (7 downto 0) := (others => '0');
                signal rect_memory_data :
        in unsigned (7 downto 0) := (others => '0');

                -- Access to line buffer
                signal screen_col : in unsigned(8 downto 0) := (others => '0');
                signal screen_col_color :
           out std_logic_vector (17 downto 0) := (others => '0');

                -- unit testing signals
                signal done : out std_logic := '0'
        );
end buffered_line_generator;
architecture caster of buffered_line_generator is
        signal write_col_offset : unsigned (3 downto 0) := (others => '0');
        constant MAX_COL_OFFSET : unsigned (3 downto 0) := "1111";
        signal current_row : signed (8 downto 0) := (others => '0');

        -- rectangle memory
        signal rec_read_x0, rec_read_y0, rec_read_z0, rec_read_x1, rec_read_y1,
rec_read_z1,
                rec_read_color : unsigned (7 downto 0) := (others => '0');
        signal rec_write_x0, rec_write_y0, rec_write_z0,
                rec_write_x1, rec_write_y1, rec_write_z1 : unsigned (8 downto 0) :=
(others => '0');
        signal rec_write_color : unsigned (5 downto 0) := (others => '0');
        signal read_rect_address : unsigned (7 downto 0) := (others => '0');
        constant MAX_REC_MEM_PLUS_1 : unsigned (7 downto 0) := "00101010";

        -- ray control signals
        type read_states is (zero, one, two, three, four, five, six, seven, eight);
```

```
	signal read_state : read_states := zero;
	type control_states is (c_zero, c_one, c_two_almost, c_two, c_three_almost,
		c_three, c_four, c_five, c_six);
	signal control_state : control_states := c_zero;
	signal ray_count : unsigned (0 downto 0) := "0";

	-- ray units
	signal new_pixel : std_logic := '0';
	signal new_rectangle : std_logic := '0';
	signal ray_units_done : std_logic := '0';
	signal done_rectangles : std_logic := '0';
	signal col_base0, col_base1, col_base2, col_base3, col_base4, col_base5,
col_base6,
		col_base7, col_base8, col_base9, col_base10, col_base11, col_base12,
col_base13,
		col_base14, col_base15 , col_base16, col_base17, col_base18,
col_base19
		: signed (8 downto 0) := (others => '0');
	signal ru_done_0, ru_done_1, ru_done_2, ru_done_3, ru_done_4, ru_done_5,
ru_done_6,
		ru_done_7, ru_done_8, ru_done_9, ru_done_10, ru_done_11, ru_done_12,
ru_done_13,
		ru_done_14, ru_done_15 , ru_done_16, ru_done_17, ru_done_18,
ru_done_19
		: std_logic := '0';
	signal ru_rgb_0, ru_rgb_1, ru_rgb_2, ru_rgb_3, ru_rgb_4, ru_rgb_5, ru_rgb_6,
ru_rgb_7,
		ru_rgb_8, ru_rgb_9, ru_rgb_10, ru_rgb_11, ru_rgb_12, ru_rgb_13,
ru_rgb_14, ru_rgb_15,
		ru_rgb_16, ru_rgb_17, ru_rgb_18, ru_rgb_19
		: unsigned (17 downto 0) := (others => '0');
	signal read_0, read_1, read_2, read_3, read_4, read_5, read_6, read_7, read_8,
read_9,
		read_10, read_11, read_12, read_13, read_14, read_15, read_16,
read_17, read_18, read_19
		: unsigned (17 downto 0) := (others => '0');

	-- pixel memory
	signal pixel_write_address, pixel_read_address : unsigned(4 downto 0) :=
"00000";
	signal write_pixels : std_logic := '0';

	constant MAX_PIXEL_BLOCK : unsigned(4 downto 0) := "10011";
	signal pixel_block : unsigned(4 downto 0) := "00000";
	signal read_col_offset : unsigned(3 downto 0) := "0000";
	signal last_screen_col : unsigned(8 downto 0) := (others => '0');
begin
	rect_memory_address <= read_rect_address;

	-- If you comment out ray units to improve compile time for debugging,
	-- make sure you update this value as well. Otherwise your architecture
	-- will just get optimized away and nothing will happen.
	ray_units_done <= ru_done_0 and ru_done_1 and ru_done_2 and ru_done_3 and
			  ru_done_4 and ru_done_5 and ru_done_6 and ru_done_7 and
			  ru_done_8 and ru_done_9 and ru_done_10 and ru_done_11 and
			  ru_done_12 and ru_done_13 and ru_done_14 and ru_done_15 and
			  ru_done_16 and ru_done_17 and ru_done_18 and ru_done_19;

	double_buffering_addresses : process(current_row, write_col_offset,
read_col_offset)
		variable even_row, odd_row : unsigned(0 downto 0) := "0";
	begin
		even_row := unsigned(current_row(0 downto 0));
		if even_row = 0 then
```

```
				pixel_write_address <= "0" & write_col_offset;
				pixel_read_address <= "1" & read_col_offset;
			else
				pixel_write_address <= "1" & write_col_offset;
				pixel_read_address <= "0" & read_col_offset;
			end if;
	end process;

	read_line_buffer : process(screen_col, read_0, read_1, read_2, read_3,
read_4, read_5, read_6,read_7, read_8, read_9,
read_10, read_11, read_12, read_13, read_14,
read_15, read_16,read_17, read_18, read_19)
			variable color : unsigned(17 downto 0) := (others => '0');
			variable block_from_col : unsigned(4 downto 0) := (others => '0');
	begin
			block_from_col := screen_col(8 downto 4);
			if block_from_col = "00000" then
				color := read_0;
			elsif block_from_col = "00001" then
				color := read_1;
			elsif block_from_col = "00010" then
				color := read_2;
			elsif block_from_col = "00011" then
				color := read_3;
			elsif block_from_col = "00100" then
				color := read_4;
			elsif block_from_col = "00101" then
				color := read_5;
			elsif block_from_col = "00110" then
				color := read_6;
			elsif block_from_col = "00111" then
				color := read_7;
			elsif block_from_col = "01000" then
				color := read_8;
			elsif block_from_col = "01001" then
				color := read_9;
			elsif block_from_col = "01010" then
				color := read_10;
			elsif block_from_col = "01011" then
				color := read_11;
			elsif block_from_col = "01100" then
				color := read_12;
			elsif block_from_col = "01101" then
				color := read_13;
			elsif block_from_col = "01110" then
				color := read_14;
			elsif block_from_col = "01111" then
				color := read_15;
			elsif block_from_col = "10000" then
				color := read_16;
			elsif block_from_col = "10001" then
				color := read_17;
			elsif block_from_col = "10010" then
				color := read_18;
			elsif block_from_col = "10011" then
				color := read_19;
			else
				color := "000000000000000000";
			end if;
			screen_col_color <= std_logic_vector(color);
			read_col_offset <= screen_col(3 downto 0);
	end process read_line_buffer;

	depending_on_control : process(clock)
```

```
	begin
		if rising_edge(clock) then
			case read_state is
			when zero =>
				read_state <= one;
			when one =>
				rec_read_x0 <= rect_memory_data;
				read_state <= two;
			when two =>
				rec_read_y0 <= rect_memory_data;
				read_state <= three;
			when three =>
				rec_read_z0 <= rect_memory_data;
				read_state <= four;
			when four =>
				rec_read_x1 <= rect_memory_data;
				read_state <= five;
			when five =>
				rec_read_y1 <= rect_memory_data;
				read_state <= six;
			when six =>
				rec_read_z1 <= rect_memory_data;
				read_state <= seven;
			when seven =>
				rec_read_color <= rect_memory_data;
				read_state <= eight;
			when eight =>
				-- hang out for a while.
			end case;
			if read_state /= seven and read_state /= eight then
				read_rect_address <= read_rect_address + 1;
			end if;

			case control_state is
			when c_zero =>
				if read_state = eight then
					control_state <= c_one;
					new_pixel <= '1';
				end if;
			when c_one => -- new rectangle ready
				new_pixel <= '0';
				done_rectangles <= '0';
				rec_write_x0 <= "0" & rec_read_x0;
				rec_write_y0 <= "0" & rec_read_y0;
				rec_write_z0 <= "0" & rec_read_z0;
				rec_write_x1 <= "0" & rec_read_x1;
				rec_write_y1 <= "0" & rec_read_y1;
				rec_write_z1 <= "0" & rec_read_z1;
				rec_write_color <= rec_read_color(5 downto 0);
				new_rectangle <= '1';
				read_state <= zero;
				if read_rect_address = MAX_REC_MEM_PLUS_1 then
					read_rect_address <= "00000000";
					control_state <= c_three_almost;
				else
					control_state <= c_two_almost;
				end if;
			when c_two_almost =>
-- just started. Don't want to check done yet, wait a cycle.
				new_rectangle <= '0';
				control_state <= c_two;
			when c_two => -- done a rectangle
				if ray_units_done = '1' and read_state = eight then
					control_state <= c_one;
```

```
                                end if;
                        when c_three_almost =>
                                new_rectangle <= '0';
                                control_state <= c_three;
                        when c_three => -- done all rectangles
                                if ray_units_done = '1' then
                                        done_rectangles <= '1';
                                        if ray_count = "1" then
                                                ray_count <= "0";
                                                control_state <= c_four;
                                        else
                                                ray_count <= ray_count + 1;
                                                control_state <= c_one;
                                        end if;
                                end if;
                        when c_four => -- done reflections
                                done_rectangles <= '0';
                                write_pixels <= '1';
                                control_state <= c_five;
                        when c_five =>
                                write_pixels <= '0';
                                if write_col_offset = MAX_COL_OFFSET then
                                        control_state <= c_six;
                                else
                                        new_pixel <= '1';
                                        write_col_offset <= write_col_offset + 1;
                                        control_state <= c_one;
                                end if;
                        when c_six => -- done row
                                done <= '1';
                        end case;
                        -- reset when the row changes
                        if current_row /= row then
                                current_row <= row;
                                new_pixel <= '1';
                                if control_state /= c_zero then
                                        control_state <= c_one;
                                end if;
                                write_col_offset <= "0000";
                                done <= '0';
                        end if;
                end if;
        end process;

        -- Obviously I couldn't get loop generate figured out.
        ray_unit_0 : entity work.ray_unit
        port map (clk => clock, col_base => col_base0, row => current_row,
col_offset => write_col_offset, cam_co_ord_x => camera_x,
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
 new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_0,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_0);
        pixel_ram_0 : entity work.pixel_ram
        port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address, di => ru_rgb_0, do => read_0);
        ray_unit_1 : entity work.ray_unit
        port map (clk => clock, col_base => col_base1, row => current_row,
 col_offset => write_col_offset,cam_co_ord_x => camera_x,
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
        new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_1,done_rectangles => done_rectangles,
```

```
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_1);
       pixel_ram_1 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,di => ru_rgb_1, do => read_1);
       ray_unit_2 : entity work.ray_unit
       port map (clk => clock, col_base => col_base2, row => current_row,
             col_offset => write_col_offset,cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
                 new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_2,done_rectangles => done_rectangles,
            rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_2);
       pixel_ram_2 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
 read_a => pixel_read_address, di => ru_rgb_2, do => read_2);
       ray_unit_3 : entity work.ray_unit
       port map (clk => clock, col_base => col_base3, row => current_row,
col_offset => write_col_offset,cam_co_ord_x => camera_x,
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
                 new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_3,done_rectangles => done_rectangles,
 rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
  rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_3);
       pixel_ram_3 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address, di => ru_rgb_3, do => read_3);
       ray_unit_4 : entity work.ray_unit
       port map (clk => clock, col_base => col_base4, row => current_row,
 col_offset => write_col_offset,cam_co_ord_x => camera_x,
    cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
         new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_4,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_4);
       pixel_ram_4 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,di => ru_rgb_4, do => read_4);
       ray_unit_5 : entity work.ray_unit
       port map (clk => clock, col_base => col_base5, row => current_row,
 col_offset => write_col_offset,cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
        new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_5,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0,rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
 rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_5);
       pixel_ram_5 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
 read_a => pixel_read_address, di => ru_rgb_5, do => read_5);
       ray_unit_6 : entity work.ray_unit
       port map (clk => clock, col_base => col_base6, row => current_row,
 col_offset => write_col_offset,cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
         new_pixel => new_pixel, new_rectangle => new_rectangle,
```

```
 done => ru_done_6,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_6);
       pixel_ram_6 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,di => ru_rgb_6, do => read_6);
       ray_unit_7 : entity work.ray_unit
       port map (clk => clock, col_base => col_base7, row => current_row,
col_offset => write_col_offset,cam_co_ord_x => camera_x,
    cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
         new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_7,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_7);
       pixel_ram_7 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,di => ru_rgb_7, do => read_7);
       ray_unit_8 : entity work.ray_unit
       port map (clk => clock, col_base => col_base8, row => current_row,
                 col_offset => write_col_offset,cam_co_ord_x => camera_x,
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
         new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_8,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_8);
       pixel_ram_8 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
  read_a => pixel_read_address, di => ru_rgb_8, do => read_8);
       ray_unit_9 : entity work.ray_unit
       port map (clk => clock, col_base => col_base9, row => current_row,
col_offset => write_col_offset, cam_co_ord_x => camera_x,
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
                 new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_9,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_9);
       pixel_ram_9 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address, di => ru_rgb_9, do => read_9);
       ray_unit_10 : entity work.ray_unit
       port map (clk => clock, col_base => col_base10, row => current_row,
col_offset => write_col_offset,cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
 new_pixel => new_pixel, new_rectangle => new_rectangle,
    done => ru_done_10,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0,rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_10);
       pixel_ram_10 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,
        di => ru_rgb_10, do => read_10);
       ray_unit_11 : entity work.ray_unit
       port map (clk => clock, col_base => col_base11, row => current_row,
    col_offset => write_col_offset, cam_co_ord_x => camera_x,
```

```
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
                 new_pixel => new_pixel, new_rectangle => new_rectangle,
     done => ru_done_11,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_11);
       pixel_ram_11 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,di => ru_rgb_11, do => read_11);
       ray_unit_12 : entity work.ray_unit
       port map (clk => clock, col_base => col_base12, row => current_row,
    col_offset => write_col_offset,cam_co_ord_x => camera_x,
cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
         new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_12,done_rectangles => done_rectangles,
 rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_12);
       pixel_ram_12 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,di => ru_rgb_12, do => read_12);
       ray_unit_13 : entity work.ray_unit
       port map (clk => clock, col_base => col_base13, row => current_row,
col_offset => write_col_offset,cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
                new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_13,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_13);
       pixel_ram_13 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address, di => ru_rgb_13, do => read_13);
       ray_unit_14 : entity work.ray_unit
       port map (clk => clock, col_base => col_base14, row => current_row,
 col_offset => write_col_offset, cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
new_pixel => new_pixel, new_rectangle => new_rectangle,
done => ru_done_14, done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
color => rec_write_color, rgb => ru_rgb_14);
       pixel_ram_14 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address,
        di => ru_rgb_14, do => read_14);
       ray_unit_15 : entity work.ray_unit
       port map (clk => clock, col_base => col_base15, row => current_row,
col_offset => write_col_offset, cam_co_ord_x => camera_x,
     cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
                 new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_15,done_rectangles => done_rectangles,
rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
 color => rec_write_color,rgb => ru_rgb_15);
       pixel_ram_15 : entity work.pixel_ram
       port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
read_a => pixel_read_address, di => ru_rgb_15, do => read_15);
       ray_unit_16 : entity work.ray_unit
```

```
        port map (clk => clock, col_base => col_base16, row => current_row,
 col_offset => write_col_offset,cam_co_ord_x => camera_x,
  cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
          new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_16,done_rectangles => done_rectangles,
 rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
     rec_y1 => rec_write_y1,rec_z1 => rec_write_z1,
  color => rec_write_color, rgb => ru_rgb_16);
        pixel_ram_16 : entity work.pixel_ram
        port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
 read_a => pixel_read_address,di => ru_rgb_16, do => read_16);
        ray_unit_17 : entity work.ray_unit
        port map (clk => clock, col_base => col_base17, row => current_row,
     col_offset => write_col_offset,cam_co_ord_x => camera_x,
      cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
         new_pixel => new_pixel, new_rectangle => new_rectangle,
  done => ru_done_17, done_rectangles => done_rectangles,
 rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
  rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
 rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_17);
        pixel_ram_17 : entity work.pixel_ram
        port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
     read_a => pixel_read_address,di => ru_rgb_17, do => read_17);
        ray_unit_18 : entity work.ray_unit
        port map (clk => clock, col_base => col_base18, row => current_row,
  col_offset => write_col_offset,cam_co_ord_x => camera_x,
  cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
 new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_18,done_rectangles => done_rectangles,
 rec_x0 => rec_write_x0, rec_y0 => rec_write_y0,
 rec_z0 => rec_write_z0, rec_x1 => rec_write_x1,
                 rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_18);
        pixel_ram_18 : entity work.pixel_ram
        port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
      read_a => pixel_read_address,di => ru_rgb_18, do => read_18);
        ray_unit_19 : entity work.ray_unit
        port map (clk => clock, col_base => col_base19, row => current_row,
  col_offset => write_col_offset, cam_co_ord_x => camera_x,
 cam_co_ord_y => camera_y, cam_co_ord_z => camera_z,
  new_pixel => new_pixel, new_rectangle => new_rectangle,
 done => ru_done_19,done_rectangles => done_rectangles,
  rec_x0 => rec_write_x0,rec_y0 => rec_write_y0,
  rec_z0 => rec_write_z0,rec_x1 => rec_write_x1,
 rec_y1 => rec_write_y1, rec_z1 => rec_write_z1,
 color => rec_write_color, rgb => ru_rgb_19);
        pixel_ram_19 : entity work.pixel_ram
        port map (clk => clock, we => write_pixels, write_a => pixel_write_address,
  read_a => pixel_read_address,di => ru_rgb_19, do => read_19);

        col_base0 <= "101100000"; col_base1 <= "101110000"; col_base2 <= "110000000";
        col_base3 <= "110010000"; col_base4 <= "110100000"; col_base5 <= "110110000";
        col_base6 <= "111000000"; col_base7 <= "111010000"; col_base8 <= "111100000";
        col_base9 <= "111110000"; col_base10 <= "000000000"; col_base11 <=
"000010000";
        col_base12 <= "000100000"; col_base13 <= "000110000"; col_base14 <=
"001000000";
        col_base15 <= "001010000"; col_base16 <= "001100000"; col_base17 <=
"001110000";
        col_base18 <= "010000000"; col_base19 <= "010010000";

end caster;
```

---

core_math.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
-- return a + b * ( c - d ) / e
-- [0 to 255] + [-80 to 79] * ( [0 to 255] - [-80 to 79]) / [-80 to 79]
entity core_math is
        port (
                signal clk : in std_logic;
                signal reset : in std_logic;
                signal a, c, d : in unsigned (8 downto 0) := (others => '0');
                signal b, e : in signed (8 downto 0) := (others => '0');
                signal result : out unsigned (8 downto 0) := (others => '0');
                signal done : out std_logic := '0' -- active high
        );
end core_math;
architecture math of core_math is
    signal done_sig : std_logic := '0';
        signal div_reset : std_logic := '0';
        signal div_done : std_logic := '0';
        signal div_result : signed (17 downto 0) := (others => '0');
        signal numerator_wire : signed (17 downto 0) := (others => '0');
        constant INFINITE : unsigned (8 downto 0) := "011111111";
        constant INFINITE_INT : integer := 255;
begin
        done <= done_sig;
        compute_core_math : process(clk)
                variable cMinusD : signed (8 downto 0) := (others => '0');
                variable numerator : signed (17 downto 0) := (others => '0');
        begin
                if rising_edge(clk) then
                        if (reset = '1') then
                                result <= "000000000";
                                if e = "000000000" then
                                        done_sig <= '1';
                                        result <= INFINITE;
                                else
                                        done_sig <= '0';
                                        cMinusD := signed(c) - signed(d);
                                        numerator := cMinusD * b;
                                        numerator_wire <= numerator;
                                        div_reset <= '1';
                                end if;
                        else
                                div_reset <= '0';
                                if div_done = '1' and done_sig = '0' then
                                        result <= INFINITE;
                                        if div_result(17) = '0' and (div_result(16) =
'1' or div_result(15) = '1') then
                                        else
                                                -- If we move backwards and the slope
is negative,
                                                -- or if we move forwards and the
slope is positive, good.
                                                -- In other words, rays can't reverse.
                                                if div_result = "000000000000000000"
or
(div_result(17) = '1' and b(8) = '1') or
(div_result(17) = '0' and b(8) = '0') then
```

```
                                                                numerator :=
signed("000000000" & a) + div_result;
                                                                if numerator(17) = '1' or
numerator(16) = '1' or numerator(15) = '1'
                                                                        or numerator(14) = '1'
or numerator(13) = '1' or numerator(12) = '1'
                                                                        or numerator(11) = '1'
or numerator(10) = '1' or numerator(9) = '1'
                                                                        or numerator(8) = '1'
then
                                                                else
                                                                        result <=
unsigned(numerator(8 downto 0));
                                                                end if;
                                                        end if;
                                                end if;
                                                done_sig <= '1';
                                        end if;
                                end if;
                        end if;
                end process;

                Divider: entity work.divider
                port map(
                        clk => clk,
                        reset => div_reset,
                        numerator => numerator_wire,
                        denominator => e,
                        result => div_result,
                        done => div_done
                );
end math;
```

---

divider.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity divider is
        port (
                clk : in std_logic;
                reset : in std_logic;

                numerator : in signed (17 downto 0) := (others => '0');
                denominator : in signed (8 downto 0) := (others => '0');

                result : out signed (17 downto 0) := (others => '0');
                done : out std_logic := '0'
        );
end entity divider;
architecture rtl of divider is
begin
        process(clk, reset)
                variable counter : unsigned (4 downto 0) := (others => '0');
                variable negative : std_logic := '0';
                variable pos_numerator : signed (17 downto 0) := (others => '0');
                variable pos_denominator : signed (8 downto 0) := (others => '0');
                variable numerator_var : unsigned (25 downto 0) := (others => '0');
                variable denominator_var : unsigned (25 downto 0) := (others => '0');
                variable result_shift : signed (17 downto 0) := (others => '0');
```

```
		variable subtract_var : signed (25 downto 0) := (others => '0');
	begin
		if reset = '1' then
			negative := '0';
			pos_numerator := numerator;
			if numerator(17) = '1' then
				negative := '1';
				pos_numerator := not(numerator) + 1;
			end if;
			pos_denominator := denominator;
			if denominator(8) = '1' then
				if negative = '1' then
					negative := '0';
				else
					negative := '1';
				end if;
				pos_denominator := not(denominator) + 1;
			end if;
			numerator_var := "00000000" & unsigned(pos_numerator);
			denominator_var := "0000" & unsigned(pos_denominator) &
"0000000000000";
			result_shift := (others => '0');
			done <= '0';

			counter := "01110";
		elsif rising_edge(clk) then
			if counter /= "00000" then
				for i in 1 to 2
				loop
					subtract_var := signed(numerator_var) -
signed(denominator_var);
					if subtract_var(25) = '1' then
						result_shift(17 downto 0) :=
result_shift(16 downto 0) & '0';
					else
						result_shift(17 downto 0) :=
result_shift(16 downto 0) & '1';
						numerator_var :=
unsigned(subtract_var);
					end if;
					denominator_var(25 downto 0) := '0' &
denominator_var(25 downto 1);
				end loop;
				counter := counter - 2;
			end if;
			if counter = "00000" then
				done <= '1';
			else
				done <= '0';
			end if;
			if negative = '1' then
				result <= not(result_shift) + 1;
			else
				result <= result_shift;
			end if;
		end if;
	end process;
end rtl;

--------------------------------------------------------------------------------
```

video_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity video_generator is
port(
        clock   : in std_logic;
        reset   : in std_logic;

        col_to_read : out unsigned (8 downto 0);
        color       : in std_logic_vector(17 downto 0);
        start_screen : out std_logic;
        start_row    : out std_logic;
        -- Signals for the vga chip
        vga_clock,
        vga_h_sync,
        vga_v_sync,
        vga_blank,
        vga_sync : out std_logic;
        vga_r,
        vga_g,
        vga_b : out unsigned(9 downto 0)
);
end video_generator;
architecture rtl of video_generator is
        -- Video parameters
        constant HTOTAL       : integer := 1600;
        constant HSYNC        : integer := 192;
        constant HBACK_PORCH  : integer := 96;
        constant HACTIVE      : integer := 1280;
        constant HFRONT_PORCH : integer := 32;

        constant VTOTAL       : integer := 525;
        constant VSYNC        : integer := 2;
        constant VBACK_PORCH  : integer := 33;
        constant VACTIVE      : integer := 480;
        constant VFRONT_PORCH : integer := 10;

        -- Stuff we found out (old, was at 25mhz):
        -- In simulation:
        -- HCount 146 is the first pixel in a row
        -- HCount 786 is the pixel after the active area
        -- We start outputting video at Vcount 35
        -- We stop at Vcount 515
        -- When we look at this on the monitor, the first column is duplicated
        -- once and last column is duplicated twice.

        -- Signals for the video controller
        signal Hcount : unsigned(10 downto 0);  -- Horizontal position
        signal Vcount : unsigned(9 downto 0);  -- Vertical position
        signal EndOfLine, EndOfField : std_logic;
        signal vga_hblank, vga_hsync,
    vga_vblank, vga_vsync : std_logic;  -- Sync. signals
        signal clock_25 : std_logic := '0';
        signal col_to_read_sig : unsigned (8 downto 0);
begin
        process (clock)
        begin
                if rising_edge(clock) then
                        clock_25 <= not clock_25;
                end if;
        end process;
```

```
-- Horizontal and vertical counters

HCounter : process (clock)
begin
	if rising_edge(clock) then
		if reset = '1' then
			Hcount <= (others => '0');
		elsif EndOfLine = '1' then
			Hcount <= (others => '0');
		else
			Hcount <= Hcount + 1;
		end if;
	end if;
end process HCounter;
EndOfLine <= '1' when Hcount = HTOTAL - 1 else '0';
VCounter : process (clock)
begin
	if rising_edge(clock) then
		if reset = '1' then
			Vcount <= (others => '0');
		elsif EndOfLine = '1' then
			if EndOfField = '1' then
				Vcount <= (others => '0');
			else
				Vcount <= Vcount + 1;
			end if;
		end if;
	end if;
end process VCounter;
EndOfField <= '1' when Vcount = VTOTAL - 1 else '0';
-- State machines to generate HSYNC, VSYNC, HBLANK, and VBLANK
HSyncGen : process (clock)
begin
	if rising_edge(clock) then
		if reset = '1' or EndOfLine = '1' then
			vga_hsync <= '1';
		elsif Hcount = HSYNC - 1 then
			vga_hsync <= '0';
		end if;
	end if;
end process HSyncGen;

HBlankGen : process (clock)
begin
	if rising_edge(clock) then
		if reset = '1' then
			vga_hblank <= '1';
		elsif Hcount = HSYNC + HBACK_PORCH then
			vga_hblank <= '0';
		elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE then
			vga_hblank <= '1';
		end if;
	end if;
end process HBlankGen;
VSyncGen : process (clock)
begin
	if rising_edge(clock) then
		if reset = '1' then
			vga_vsync <= '1';
		elsif EndOfLine ='1' then
			if EndOfField = '1' then
				vga_vsync <= '1';
			elsif Vcount = VSYNC - 1 then
				vga_vsync <= '0';
```

```
                                end if;
                        end if;
                end if;
        end process VSyncGen;
        VBlankGen : process (clock)
        begin
                if rising_edge(clock) then
                        if reset = '1' then
                                vga_vblank <= '1';
                        elsif EndOfLine = '1' then
                                if Vcount = VSYNC + VBACK_PORCH - 1 then
                                        vga_vblank <= '0';
                                elsif Vcount = VSYNC + VBACK_PORCH + VACTIVE - 1 then
                                        vga_vblank <= '1';
                                end if;
                        end if;
                end if;
        end process VBlankGen;

        -- Start screen and start row control signals
        start_signals : process (clock)
        begin
                if rising_edge(clock) then
                        if reset = '1' then
                                start_screen <= '0';
                                start_row <= '0';
                        -- We always want to assert these signals just as soon as
we're
                        -- done with the active area of the previous row.
                        elsif Hcount = HSYNC + HBACK_PORCH + HACTIVE + 1 then
                                if Vcount = VSYNC + VBACK_PORCH - 1 then
                                        start_screen <= '1';
                                else
                                        start_screen <= '0';
                                end if;
                                -- only start a new row of ray tracing every other
screen row
                                -- to expand 240 rows of ray traced data 480 rows on
screen.
                                if Vcount(0) = '0'
                                   and Vcount >= VSYNC + VBACK_PORCH
                                   and Vcount < VSYNC + VBACK_PORCH + VACTIVE then
                                        start_row <= '1';
                                else
                                        start_row <= '0';
                                end if;
                        else
                                start_screen <= '0';
                                start_row <= '0';
                        end if;
                end if;
        end process start_signals;

        -- Column that we want to read from the line buffer
        column : process (clock)
        begin
                if rising_edge(clock) then
                        if reset = '1' then
                                col_to_read_sig <= (others => '0');
                        elsif Hcount = HSYNC + HBACK_PORCH then
                                col_to_read_sig <= (others => '0');
                        -- only increase the col every other pixel to expand 320
pixels in
                        -- memory to 640 pixels on screen.
```

```
                        -- divide by another factor of two since hcount is now running
at
                        -- twice the vga clock speed.
                        elsif Hcount(1 downto 0) = "00" then
                                col_to_read_sig <= col_to_read_sig + "000000001";
                        end if;
                end if;
        end process column;

        col_to_read <= col_to_read_sig;
        -- Final video out
        -- TODO do we need to ensure we always send out white so it gets the right
        -- levels or something?
        VideoOut : process (clock, reset)
        begin
                if reset = '1' then
                        vga_r <= "0000000000";
                        vga_g <= "0000000000";
                        vga_b <= "0000000000";
                elsif rising_edge(clock) then
                        if vga_hblank = '1' or vga_vblank = '1' then
                                vga_r <= "0000000000";
                                vga_g <= "0000000000";
                                vga_b <= "0000000000";
                        else
                                vga_r <= unsigned(color(17 downto 12)) & "0000";
                                vga_g <= unsigned(color(11 downto 6)) & "0000";
                                vga_b <= unsigned(color(5 downto 0)) & "0000";
                        end if;
                end if;
        end process VideoOut;

        vga_clock <= clock_25;
        vga_h_sync <= not vga_hsync;
        vga_v_sync <= not vga_vsync;
        vga_sync <= '0';
        vga_blank <= not (vga_hsync or vga_vsync);
end rtl;
```

----

pixel_ram.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pixel_ram is
        port (
                clk : in std_logic := '0';
                we : in std_logic := '0'; -- write enabled
                write_a, read_a : in unsigned(4 downto 0) := (others => '0');
                di : in unsigned(17 downto 0) := (others => '0'); -- data in
                do : out unsigned(17 downto 0) := (others => '0') -- data out
        );
end pixel_ram;
architecture rtl of pixel_ram is
        type ram_type is array (31 downto 0) of unsigned(17 downto 0);
        signal RAM : ram_type := (others => "000000000000000000");
        signal read_a_s : unsigned(4 downto 0) := (others => '0');
begin
        process (clk)
        begin
```

```
            if clk = '1' then
                    if we = '1' then
                            RAM(to_integer(write_a)) <= di;
                    end if;
                    read_a_s <= read_a;
            end if;
    end process;

    do <= RAM(to_integer(read_a_s));
end rtl;
```

----

pixel_ram_tester.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity pixel_ram_tester is
end pixel_ram_tester;
architecture pr_test of pixel_ram_tester is
        signal clk : std_logic := '0';
        signal we :     std_logic := '0';                       -- write enabled
        signal write_a, read_a : unsigned(4 downto 0) := (others => '0');
        signal di :     unsigned(17 downto 0) := (others => '0'); -- data in
        signal do :     unsigned(17 downto 0) := (others => '0'); -- data out
        constant VAL_1 : unsigned(17 downto 0) := "101010001010101001";
        constant VAL_2 : unsigned(17 downto 0) := "110101001000001010";
        constant VAL_3 : unsigned(17 downto 0) := "000101010101001111";
        constant VAL_4 : unsigned(17 downto 0) := "111111111000000000";
        constant VAL_5 : unsigned(17 downto 0) := "000101010101111101";
begin
        process
        begin
                loop
                        wait for 10 ns;
                        clk <= '1';
                        wait for 10 ns;
                        clk <= '0';
                end loop;
        end process;

        pixel_ram_instance : entity work.pixel_ram port map (
                clk => clk,
                we => we,
                write_a => write_a,
                read_a => read_a,
                di => di,
                do => do
        );

        process
        begin
                write_a <= "00000"; di <= VAL_1; we <= '1';
                wait for 20 ns;
                we <= '0'; read_a <= "00000";
                wait for 20 ns;
                assert do = VAL_1 report "First" severity error;
                write_a <= "00001"; di <= VAL_2; we <= '1';
                wait for 20 ns;
                write_a <= "00010"; di <= VAL_3;
                wait for 20 ns;
```

```
                write_a <= "00011"; di <= VAL_4; read_a <= "00010";
                wait for 20 ns;
                we <= '0';
                assert do = VAL_3 report "Second" severity error;
                read_a <= "00000";
                wait for 20 ns;
                assert do = VAL_1 report "Third" severity error;
                report "NONE. End of simulation." severity failure;
        end process;
 end pr_test;
```

----

ray_unit_tester.vhd - this was generated by the c prototype.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ray_unit_tester is
end ray_unit_tester;

architecture ray_test of ray_unit_tester is
    signal clk : std_logic := '0';
    signal col_base : signed (8 downto 0) := (others => '0');

  -- should be one of these: -160, -150, ... , -10, 0, 10, ... , 150
    signal row : signed (8 downto 0) := (others => '0'); -- should start at 189 and go
down to -50
    signal col_offset : unsigned (3 downto 0) := (others => '0'); -- should start at 0
and go to 9
    signal cam_co_ord_x, cam_co_ord_y, cam_co_ord_z : unsigned (8 downto 0) := (others
=> '0');

  -- coordinates of the camera

    -- control signals
    signal new_pixel : std_logic := '0'; -- assert for 1 cycle when starting a new
pixel
    signal new_rectangle : std_logic := '0'; -- assert for 1 cycle with each new
rectangle
    signal done : std_logic := '0'; -- active high. Wait for this signal after you
assert new_rectangle
    signal done_rectangles : std_logic := '0'; -- assert for 1 cycle when you finish
all the rectangles

    -- rectangles are 7 bytes. x,y,z for corners 1 and 2, and 1 byte for color.
    signal rec_x0, rec_y0, rec_z0, rec_x1, rec_y1, rec_z1 : unsigned (8 downto 0) :=
(others => '0');
    signal color : unsigned (5 downto 0) := (others => '0');

    -- color output
    -- this is the final result.
    signal red, green, blue : unsigned (5 downto 0) := (others => '0');
begin
    unit: entity work.ray_unit port map (
        clk => clk,
        col_base => col_base,
        row => row,
        col_offset => col_offset,
        cam_co_ord_x => cam_co_ord_x,
```

```
        cam_co_ord_y => cam_co_ord_y,
        cam_co_ord_z => cam_co_ord_z,
        new_pixel => new_pixel,
        new_rectangle => new_rectangle,
        done => done,
        done_rectangles => done_rectangles,
        rec_x0 => rec_x0,
        rec_y0 => rec_y0,
        rec_z0 => rec_z0,
        rec_x1 => rec_x1,
        rec_y1 => rec_y1,
        rec_z1 => rec_z1,
        color => color,
        red => red,
        green => green,
        blue => blue
    );

    process
    begin
        loop
            wait for 10 ns;
            clk <= '1';
            wait for 10 ns;
            clk <= '0';
        end loop;
    end process;

    process
    begin
        cam_co_ord_x <= "000000000"; cam_co_ord_y <= "010000000"; cam_co_ord_z <=
"000011001";
        col_base <= "101100000"; row <= "111011110"; col_offset <= "0001";
        new_pixel <= '1'; wait for 20 ns; new_pixel <= '0';
        rec_x0 <= "000000000"; rec_y0 <= "000000000"; rec_z0 <= "000000000";
        rec_x1 <= "011001000"; rec_y1 <= "011001000"; rec_z1 <= "000000000";
        color <= "111100"; new_rectangle <= '1'; wait for 20 ns;
        new_rectangle <= '0'; wait until done = '1'; wait for 20 ns;
        rec_x0 <= "000110010"; rec_y0 <= "000000000"; rec_z0 <= "000000000";
        rec_x1 <= "000110010"; rec_y1 <= "001100100"; rec_z1 <= "011111110";
        color <= "110000"; new_rectangle <= '1'; wait for 20 ns;
        new_rectangle <= '0'; wait until done = '1'; wait for 20 ns;
        done_rectangles <= '1'; wait for 20 ns; done_rectangles <= '0';
        rec_x0 <= "000000000"; rec_y0 <= "000000000"; rec_z0 <= "000000000";
        rec_x1 <= "011001000"; rec_y1 <= "011001000"; rec_z1 <= "000000000";
        color <= "111100"; new_rectangle <= '1'; wait for 20 ns;
        new_rectangle <= '0'; wait until done = '1'; wait for 20 ns;
        rec_x0 <= "000110010"; rec_y0 <= "000000000"; rec_z0 <= "000000000";
        rec_x1 <= "000110010"; rec_y1 <= "001100100"; rec_z1 <= "011111110";
        color <= "110000"; new_rectangle <= '1'; wait for 20 ns;
        new_rectangle <= '0'; wait until done = '1'; wait for 20 ns;
        done_rectangles <= '1'; wait for 20 ns; done_rectangles <= '0';
        assert red = "011111" report "red 011111 (-159,-34) or (319,224)" severity
error;
        assert green = "011111" report "green 011111 (-159,-34) or (319,224)" severity
error;
        assert blue = "000101" report "blue 000101 (-159,-34) or (319,224)" severity
error;

        -- Repeat with 300 more tests.
        -- About 7,000 lines later ...

        report "Don't worry, it's just the end of the simulation." severity failure;
    end process;
```

```
end ray_test;
```

----

divider_tester.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity divider_tester is
end divider_tester;
architecture test of divider_tester is
        signal clk : std_logic := '0';
        signal reset : std_logic := '0';
        signal numerator : signed(17 downto 0) := (others => '0');
        signal denominator : signed(8 downto 0) := (others => '0');
        signal result : signed(17 downto 0) := (others => '0');
        signal done : std_logic := '0';
begin
        divider_instance: entity work.divider port map (
                clk => clk,
                reset => reset,
                numerator => numerator,
                denominator => denominator,
                result => result,
                done => done
        );
        process
        begin
                loop
                        wait for 10 ns;
                        clk <= '1';
                        wait for 10 ns;
                        clk <= '0';
                end loop;
        end process;

        process
        begin
                numerator <= "00" & x"0001";
                denominator <= "0" & x"01";
                reset <= '1'; wait for 20 ns;
                reset <= '0'; wait for 400 ns;
                assert result = "00" & x"0001" report "1 / 1 wrong" severity error;

                numerator <= "00" & x"0010";
                denominator <= "0" & x"10";
                reset <= '1'; wait for 20 ns;
                reset <= '0'; wait for 400 ns;
                assert result = "00" & x"0001" report "16 / 16 wrong" severity error;
                numerator <= "00" & x"0010";
                denominator <= "0" & x"08";
                reset <= '1'; wait for 20 ns;
                reset <= '0'; wait for 400 ns;
                assert result = "00" & x"0002" report "16 / 8 wrong" severity error;
                numerator <= "00" & x"0010";
                denominator <= "0" & x"03";
                reset <= '1'; wait for 20 ns;
                reset <= '0'; wait for 400 ns;
                assert result = "00" & x"0005" report "16 / 3 wrong" severity error;
                numerator <= "00" & x"0344";
                denominator <= "0" & x"2D";
```

```
                reset <= '1'; wait for 20 ns;
                reset <= '0'; wait for 400 ns;
                assert result = "00" & x"0012" report "836 / 45 wrong" severity error;
                report "NONE. End of simulation." severity failure;
        end process;
end test;
```

RayTracer.c - C prototype

```
#include <stdio.h>
#include <stdlib.h>
#include "vision_utilities/vision_utilities.c"
/*** The Scene ***/
const int RECTANGLES = 6;
const int ROWS = 240;
const int COLS = 320;
const int CAMERA_X_SLOPE = 50;
const int CAMERA_X = 231, CAMERA_Y = 171, CAMERA_Z = 10;
const int INFINITE = 255;

//          Rectangle     0        1          2           3        4        5          6
7     8
int recCorner1X[] = {0,        254,     241,    241,     241,     200,    125,125,
125};
int recCorner1Y[] = {0,        0,         161,    161,     161,     50,      75,  75,
75};
int recCorner1Z[] = {0,      0,          1,          11,     21,      254,    50,  50,
50};

int recCorner2X[] = {254,    254,     251,    251,     251,     200,    125,150, 150};
int recCorner2Y[] = {254,    254,     181,    181,     181,     200,     50,  50, 75};
int recCorner2Z[] = {0,       254,     1,         11,     21,      251,     75,  50,
75};

int recRed[] =         {255,     0,         0,         0,         0,       255,    255,
0, 255};
int recGreen[] =    {255,     0,       255,    255,    255,        0,        0,  255,
255};
int recBlue[] =        {0,       255,       0,         0,         0,       255,
255,255, 255};
int forInverting[9];
void IntToArray(int toSet) {
    int mult = 256; int index = 0;
    for (index = 0; index < 9; index ++) {
        if (toSet >= mult) {
            forInverting[index] = 1;
            toSet -= mult;
        } else {
            forInverting[index] = 0;
        }
        mult /= 2;
    }
}
void PrintIntInBinary(int toPrint, int digits) {
    int index = 0;
    if (toPrint < 0) {
        toPrint *= -1;
        IntToArray(toPrint - 1);
        for (index = 0; index < 9; index ++) {
            forInverting[index] = (forInverting[index] + 1) % 2;
```

```
        }
    } else {
        IntToArray(toPrint);
    }
    for (index = 9 - digits; index < 9; index ++) {
        printf("%d", forInverting[index]);
    }
}
/*** Intersection ***/
int MathCore(int start, int slope, int known, int startKnown, int knownSlope) {
    int coreValue = 10;
    if (knownSlope == 0) {
        coreValue = INFINITE;
    } else {
        // uncomment for interesting effects
        int returnValue = (start + slope * (known - startKnown) / knownSlope); // %
INFINITE;
        if (returnValue > INFINITE || returnValue < 0) {
            coreValue = INFINITE;
        } else if ((slope <= 0 && returnValue <= start) || (slope >= 0 && returnValue
>= start)) {
            coreValue = returnValue;
        } else {
            coreValue = INFINITE;
        }
    }
    return coreValue;
}
int red, green, blue; // color of intersection
int intersectX, intersectY, intersectZ; // point of intersection
int reflectX, reflectY, reflectZ; // direction of reflection
void SetIntersection(int xStart, int yStart, int zStart, int xSlope, int ySlope, int
zSlope) {
    blue = 0;
    red = 0; green = 0;
    if (zSlope >= -50) {
        blue = zSlope + 50; // Background color
    }
    intersectX = INFINITE; intersectY = INFINITE; intersectZ = INFINITE;
    reflectX = xSlope; reflectY = ySlope; reflectZ = zSlope;
    int closestDistance = INFINITE, nextDistance;
    int x, y, z; // where the ray intersects the plane through this rectangle
    int rec;
    for (rec = 0; rec < RECTANGLES; rec ++) {
        if (recCorner1X[rec] == recCorner2X[rec]) {
            x = recCorner1X[rec];
            y = MathCore(yStart, ySlope, x, xStart, xSlope);
            z = MathCore(zStart, zSlope, x, xStart, xSlope);
        } else if (recCorner1Y[rec] == recCorner2Y[rec]) {
            y = recCorner1Y[rec];
            x = MathCore(xStart, xSlope, y, yStart, ySlope);
            z = MathCore(zStart, zSlope, y, yStart, ySlope);
        } else if (recCorner1Z[rec] == recCorner2Z[rec]) {
            z = recCorner1Z[rec];
            x = MathCore(xStart, xSlope, z, zStart, zSlope);
            y = MathCore(yStart, ySlope, z, zStart, zSlope);
        } else {
            printf("Rectangle %d is not parallel to x = 0, y = 0, or z = 0", rec);
            exit(1);
        }
        // Is the intersection inside our coordinate system?
        if (x >= 0 && x < INFINITE && y >= 0 && y < INFINITE && z >= 0 && z < INFINITE
&& x != xStart) {
            // Is the intersection actually inside the rectangle?
```

```
            if (
                    (
                        (x <= recCorner1X[rec] && x >= recCorner2X[rec]) ||
                        (x >= recCorner1X[rec] && x <= recCorner2X[rec])
                    ) && (
                        (y <= recCorner1Y[rec] && y >= recCorner2Y[rec]) ||
                        (y >= recCorner1Y[rec] && y <= recCorner2Y[rec])
                    ) && (
                        (z <= recCorner1Z[rec] && z >= recCorner2Z[rec]) ||
                        (z >= recCorner1Z[rec] && z <= recCorner2Z[rec])
                    )
                ) {
                // Is the intersection closer than the previous closest?
                if (x < xStart) {
                    nextDistance = xStart - x;
                } else {
                    nextDistance = x - xStart;
                }
                if (nextDistance < closestDistance) {
                    closestDistance = nextDistance;
                    red = recRed[rec]; green = recGreen[rec]; blue = recBlue[rec];
                    intersectX = x; intersectY = y; intersectZ = z;
                    reflectX = xSlope; reflectY = ySlope; reflectZ = zSlope;
                    if (recCorner1X[rec] == recCorner2X[rec]) {
                        reflectX *= -1;
                    } else if (recCorner1Y[rec] == recCorner2Y[rec]) {
                        reflectY *= -1;
                    } else if (recCorner1Z[rec] == recCorner2Z[rec]) {
                        reflectZ *= -1;
                    }
                }
            }
        }
    }
}
void PrintSingleCoord(int value, int location) {
    // for buffered_line_generator_tester.vhd
    /* printf("\t\telsif rect_memory_address_delayed = \"");
    PrintIntInBinary(location, 8);
    printf("\" then\n\t\t\trect_memory_data <= \"");
    PrintIntInBinary(value, 8);
    printf("\";\n"); */
    // for ray_tracer_avalon_module_tester.vhd
    printf("\t\taddress <= \"");
    PrintIntInBinary(location + 3, 7);
    printf("\"; writedata <= \"00000000");
    PrintIntInBinary(value, 8);
    printf("\"; write <= '1'; wait for 20 ns; write <= '0'; wait for 20 ns;\n");
}
/*** Run the ray trace ***/
void PrintRecCoord(int index) {
    int baseLocation = 7 * index;
    PrintSingleCoord(recCorner1X[index], baseLocation);
    PrintSingleCoord(recCorner1Y[index], baseLocation + 1);
    PrintSingleCoord(recCorner1Z[index], baseLocation + 2);
    PrintSingleCoord(recCorner2X[index], baseLocation + 3);
    PrintSingleCoord(recCorner2Y[index], baseLocation + 4);
    PrintSingleCoord(recCorner2Z[index], baseLocation + 5);
    int colorValue = 0;
    if (recRed[index] == 255) { colorValue += 3 * 4 * 4; }
    if (recGreen[index] == 255) { colorValue += 3 * 4; }
    if (recBlue[index] == 255) { colorValue += 3; }
    PrintSingleCoord(colorValue, baseLocation + 6);
}
```

```
const int TEST_PIXEL_SPACE = 29;
int main() {
    ImageColor newImg;
    int rec, col, row;
    setSizeColor(&newImg, ROWS, COLS);
    setColorsColor(&newImg, 255);

    int imageRow, imageColumn;
    int currentRed, currentGreen, currentBlue;
    int shouldPrint = 0;

    int i;
    for (i = 0; i < 6; i ++) {
        PrintRecCoord(i);
    }

    printf("\n\t\tcamera_x => \"");
    PrintIntInBinary(CAMERA_X, 9);
    printf("\", camera_y => \"");
    PrintIntInBinary(CAMERA_Y, 9);
    printf("\", camera_z => \"");
    PrintIntInBinary(CAMERA_Z, 9);
    printf("\",\n\n");

    for (row = ROWS - 50; row > -50; row --) {
        imageRow = ROWS - (row + 50);
        shouldPrint = 0;
        /* printf("\t\twait until done = '1'; row <= \"");
        PrintIntInBinary(row - 1, 9);
        printf("\";\n");*/
        if (imageRow % 10 == 0 && imageRow != 0) {
            shouldPrint = 1;
        }
        for (col = - COLS / 2; col < COLS / 2; col ++) {
            imageColumn = col + COLS / 2;
            SetIntersection(CAMERA_X, CAMERA_Y, CAMERA_Z, CAMERA_X_SLOPE, col, row);
            currentRed = red / 2; currentGreen = green / 2; currentBlue = blue / 2;
            // better saturation
            // currentRed = 3 * red / 4; currentGreen = 3 * green / 4; currentBlue = 3
* blue / 4;
            // 1 reflection
            SetIntersection(intersectX, intersectY, intersectZ, reflectX, reflectY,
reflectZ);
            currentRed += red / 4; currentGreen += green / 4; currentBlue += blue / 4;
            currentRed = 4 * (currentRed / 4);
            currentGreen = 4 * (currentGreen / 4);
            currentBlue = 4 * (currentBlue / 4);
            setPixelColor(&newImg, imageRow, imageColumn, currentRed, currentGreen,
currentBlue);
            /* if (shouldPrint == 1) {
                printf("\t\tscreen_col_to_read <= \"");
                PrintIntInBinary(imageColumn, 9);
                printf("\"; wait for 40 ns;\n");
                if (imageColumn > 0) {
                    printf("\t\tassert screen_col_color = \"");
                    PrintIntInBinary(currentRed / 4, 6);
                    PrintIntInBinary(currentGreen / 4, 6);
                    PrintIntInBinary(currentBlue / 4, 6);
                    printf("\" report \"row %d, col %d r,g,b = %d, %d, %d\" severity
error;\n",
                        row, col, currentRed / 4, currentGreen / 4, currentBlue / 4);

                }
            }*/
```

```
        }
    }
    printf("\t\twait until done = '1';\n");
    writeImageColor(&newImg, "RayTrace.pmg");

    for (i = 0; i < 30; i ++) {
        printf("\t\telsif screen_col < %d then\n\t\tread_c_o_temp
:= screen_col - %d;\n\t\tcolor := read_%d;\n", (i + 1) * 11, i * 11, i);
    }

    for (i = 0; i < 30; i ++) {
        printf("col_base%d <= \"", i);
        PrintIntInBinary(i * 11 - 160, 9);
        printf("\"; ", i);
    }
    return 0;
}
```

----

main.c - Control program which ran on the nios

```
#include <stdio.h>
#include <system.h>
#include <io.h>
#include "altera_ps2/alt_up_ps2_port.h"
#include "altera_ps2/ps2_keyboard.h"
#include "raytracer.h"
#include "simulations.h"

void DontCrash();

/**
 * Returns 0 on success, 1 on failure.
 **/
int init_keyboard() {
    PS2_DEVICE ps2_mode;

    printf("Initializing keyboard.\n");
    clear_FIFO();
    ps2_mode = get_mode();
    if (ps2_mode != PS2_KEYBOARD) {
        printf("Error: Didn't detect keyboard.\n");
        return 1;
    }
    return 0;
}

int simulation = 0;

alt_u8 LatestKey() {
    KB_CODE_TYPE kb_decode_mode;
    alt_u8 key;
    int ps2_status;
    ps2_status = read_make_code_no_block(&kb_decode_mode, &key);
    if (ps2_status == PS2_NOT_READY) {
        return 0x00;
    } else if (ps2_status == PS2_ERROR) {
        printf("Error reading from ps2 port.\n");
        return 0x00;
    } // If we get here, ps2_status == PS2_SUCCESS
    if (KB_1_DOWN == key) {
```

```
        simulation = HOT_POTATO;
    } else if (KB_2_DOWN == key) {
        simulation = BOUNCING_SQUARE;
    } else if (KB_3_DOWN == key) {
        simulation = FLASHING_CUBES;
    } else if (KB_4_DOWN == key) {
        simulation = DRIVING;
    } else if (KB_ESC_DOWN == key) {
        simulation = DONE;
    } else if (KB_5_DOWN == key) {
        simulation = DONT_CRASH;
    }
    printf("Got key %x\n", key);
    return key;
}

int direction(current, min, max, location) {
    if ((location <= min && current < 0) || (location >= max && current > 0)) {
        current *= -1;
    }
    return current;
}

void BouncingSquare() {
    write_rect(0, 0, 0, 0, 254, 254, 0, rgb_to_color(3, 3, 3));
    int thirds = 0, move = 0, left = 0, left_direction = 1, front = 0, front_direction
= 1, top = 0, top_direction = 1;
    int rectangleWidth = 50;
    while (BOUNCING_SQUARE == simulation) {
        LatestKey();
        if (thirds == 0) {
            write_rect(1, front, left, top, front, rectangleWidth + left,
rectangleWidth + top, rgb_to_color(3, 0, 0));
            write_rect(2, rectangleWidth + front, left, top, rectangleWidth + front,
rectangleWidth + left, rectangleWidth + top, rgb_to_color(0, 3, 0));
        } else if (thirds == 1) {
            write_rect(1, front, left, top, rectangleWidth + front, rectangleWidth +
left, top, rgb_to_color(0, 0, 3));
            write_rect(2, front, left, rectangleWidth + top, rectangleWidth + front,
rectangleWidth + left, rectangleWidth + top, rgb_to_color(3, 3, 0));
        } else if (thirds == 2) {
            write_rect(1, front, left, top, rectangleWidth + front, left,
rectangleWidth + top, rgb_to_color(3, 0, 3));
            write_rect(2, front, rectangleWidth + left, top, rectangleWidth + front,
rectangleWidth + left, rectangleWidth + top, rgb_to_color(0, 3, 3));
        }
        if (move % 250 == 0) {
            left_direction = direction(left_direction, 0, 250 - rectangleWidth, left);
            left += left_direction;
        }
        if (move % 200 == 0) {
            front_direction = direction(front_direction, 0, 250 - rectangleWidth,
front);
            front += front_direction;
        }
        if (move % 100 == 0) {
            top_direction = direction(top_direction, 0, 250 - rectangleWidth, top);
            top += top_direction;
        }
        move = (move + 1) % 500;
        thirds = (thirds + 1) % 3;
    }

}
```

```
void FlashingColumns() {
    write_rect(0, 75, 0, 0, 75, 254, 254, rgb_to_color(0, 0, 0));
    int fourths = 0;
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_X, 5);
    int camera_c = 0;
    int camera_x = 0, camera_dx = 1, camera_y = 0, camera_dy = 1,
        camera_z = 0, camera_dz = 1;
    while (FLASHING_CUBES == simulation) {
        LatestKey();
        camera_c = (camera_c + 1) % 500;
        if (0 == camera_c) {
            camera_dx = direction(camera_dx, 0, 74, camera_x);
            camera_x += camera_dx;
            camera_dy = direction(camera_dy, 0, 254, camera_y);
            camera_y += camera_dy;
            camera_dz = direction(camera_dz, 0, 201, camera_z);
            camera_z += camera_dz;
            IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_X, camera_x);
            IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Y, camera_y);
            IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Z, camera_z);
        }
        fourths = (fourths + 1) % 4;
        if (0 == fourths) {
            write_rect(1, 40, 25, 0, 40, 50, 250, rgb_to_color(3, 3, 0));
            write_rect(2, 40, 225, 0, 40, 200, 250, rgb_to_color(3, 0, 3));
        } else if (1 == fourths) {
            write_rect(1, 65, 25, 0, 65, 50, 250, rgb_to_color(3, 3, 0));
            write_rect(2, 65, 225, 0, 65, 200, 250, rgb_to_color(3, 0, 3));
        } else if (2 == fourths) {
            write_rect(1, 40, 50, 0, 65, 50, 250, raw_rgb_to_color(3, 0, 0));
            write_rect(2, 40, 200, 0, 65, 200, 250, raw_rgb_to_color(0, 3, 0));
        } else if (3 == fourths) {
            write_rect(1, 40, 25, 0, 65, 25, 250, raw_rgb_to_color(3, 0, 0));
            write_rect(2, 40, 225, 0, 65, 225, 250, raw_rgb_to_color(0, 3, 0));
        }
    }
}

void Driving() {
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Z, 10);
    write_rect(0, 0, 0, 0, 254, 254, 0, rgb_to_color(3, 3, 0));
    write_rect(1, 254, 0, 0, 254, 254, 254, rgb_to_color(0, 0, 3));
    int camera_c = 0;
    int camera_x = 230, camera_y = 170, camera_dy = 1;
    int ship_x1, ship_x2, ship_z = 1, ship_dz = 1;
    while (DRIVING == simulation) {
        LatestKey();
        camera_c = (camera_c + 1) % 4000;
        if (0 == camera_c) {
            ship_dz = direction(ship_dz, 1, 10, ship_z);
            ship_z += ship_dz;
        }
        if (0 == camera_c % 1000) {
            camera_dy = direction(camera_dy, 11, 245, camera_y);
            camera_y += camera_dy;
            IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Y, camera_y);
        }
        if (0 == camera_c % 500) {
            camera_x = (camera_x + 1) % 254;
            IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_X, camera_x);
            ship_x1 = camera_x + 10;
            if (ship_x1 > 254) { ship_x1 = 254; }
            ship_x2 = camera_x + 20;
```

```
            if (ship_x2 > 254) { ship_x2 = 254; }
            write_rect(2, ship_x1, camera_y - 10, ship_z, ship_x2, camera_y + 10,
ship_z, rgb_to_color(0, 3, 0));
            write_rect(3, ship_x1, camera_y - 10, ship_z + 10, ship_x2, camera_y + 10,
ship_z + 10, rgb_to_color(0, 3, 0));
            write_rect(4, ship_x1, camera_y - 10, ship_z + 20, ship_x2, camera_y + 10,
ship_z + 20, rgb_to_color(0, 3, 0));
            write_rect(5, 200, 50, 254, 200, 200, ship_x2, rgb_to_color(3, 0, 3));
        }
    }
}

int Abs(int value) {
    if (value < 0) { return - value; }
    return value;
}

int ClosestValid(int coordinate) {
    if (coordinate < 0) {
        return 0;
    } else if (coordinate > 254) {
        return 254;
    } else {
        return coordinate;
    }
}

int recX1[] = {11, 250, 75,  10 }, recX2[] = {0, 250};
int recY1[] = {128,120, 125, 100}, recY2[] = {0, 140};
int recZ1[] = {1,  120, 175, 200}, recZ2[] = {1, 140};

int recR[] = {0, 3};
int recG[] = {0, 0};
int recB[] = {3, 0};

void RenderShipAim() {
    int i;
    for (i = 0; i < 2; i ++) {
        write_rect(i,
            ClosestValid(recX1[i]), ClosestValid(recY1[i]), ClosestValid(recZ1[i]),
            ClosestValid(recX2[i]), ClosestValid(recY2[i]), ClosestValid(recZ2[i]),
            raw_rgb_to_color(recR[i], recG[i], recB[i]));
    }
}

int MoveFromD(int distance) {
    if (distance > 0) {
        return 1;
    } else if (distance < 0) {
        return -1;
    }
    return 0;
}

int dest_y = 128;
int dest_z = 0;
void MoveHotPotatoShip() {
    recX1[0] = ClosestValid(recX1[0] + 1);
    if (254 <= recX1[0]) {
        recX1[0] = 0;
    }
    recY1[0] += MoveFromD(dest_y - recY1[0]);
    int z_dist = dest_z - recZ1[0];
    recZ1[0] += MoveFromD(z_dist);
```

```
    if (0 == z_dist && dest_z > 0) {
        dest_z = 0;
    }
    recX2[0] = recX1[0] + 20;
    recY2[0] = recY1[0] + 20;
    recZ2[0] = recZ1[0];
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_X, ClosestValid(recX1[0] -
20));
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Y, ClosestValid(recY1[0] +
10));
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Z, ClosestValid(recZ1[0] +
10));
}

const int MAX_MOVEMENT = 25;
int movement[] = {
    1, 0, 0, 0, 1, 0, 0, 1,
    0, 1, 0, 1, 1, 1, 0, 1, 0,
    1, 0, 0, 1, 0, 0, 0, 1};

int a_dy = 0, a_ddy = 0;
int a_dz = 0, a_ddz = 0;
void MoveAimer() {
    if (a_ddy > 0) {
        recY1[1] += a_dy * movement[a_ddy];
        recY2[1] = recY1[1] + 20;
        a_ddy --;
    }
    if (a_ddz > 0) {
        recZ1[1] += a_dz * movement[a_ddz];
        recZ2[1] = recZ1[1] + 20;
        a_ddz --;
    }
}

int movingBack[] = {0, 0, 0};
int aimX[] = {0, 0, 0};
int aimY[] = {0, 0, 0};
void MoveBullets() {
    int b;
    for (b = 0; b < 2; b ++) {
        if (1 == movingBack[b]) {
            int xs = 250 - recX1[2 + b];
            int ys = recY1[1] - recY1[2 + b];
            int zs = recZ1[1] - recZ1[2 + b];
            if (xs == 0 && ys == 0 && zs == 0) {
                aimX[b] = ClosestValid(recX1[0] + 50);
                aimY[b] = recY1[1];
                movingBack[b] = 0;
            } else {
                recX1[2 + b] += MoveFromD(xs);
                recY1[2 + b] += MoveFromD(ys);
                recZ1[2 + b] += MoveFromD(zs);
            }
        } else {
            recX1[2 + b] += MoveFromD(aimX[b] - recX1[2 + b]);
            recY1[2 + b] += MoveFromD(aimY[b] - recY1[2 + b]);
            recZ1[2 + b] += MoveFromD(0 - recZ1[2 + b]);
        }
    }
}

void NewY(int newY) {
    dest_y = newY;
```

```
    dest_z = Abs(recY1[0] - dest_y) / 2;
}

int nextBullet = 0;
void HandleKey() {
    alt_u8 key = LatestKey();
    if (0x29 == key) { // space fire
        int nb, found = 0;
        for (nb = 0; nb < 3; nb ++) {
            int anb = (nextBullet + nb) % 3;
            if (0 == found && 0 == movingBack[anb]) {
                found = 1;
                movingBack[anb] = 1;
                nextBullet = (nextBullet + 1) % 3;
            }
        }
    } else {
        if (0x41 == key) { // a p1
            NewY(20);
        } else if (0x53 == key) { // s p1
            NewY(70);
        } else if (0x44 == key) { // d p1
            NewY(125);
        } else if (0x46 == key) { // f p1
            NewY(175);
        } else if (0x47 == key) { // g p1
            NewY(230);
        } else if (0x4a == key) { // j p2 left
            a_dy = -1; a_ddy = MAX_MOVEMENT;
        } else if (0x49 == key) { // i p2 uip
            a_dz = 1; a_ddz = MAX_MOVEMENT;
        } else if (0x4b == key) { // k p2 down
            a_dz = -1; a_ddz = MAX_MOVEMENT;
        } else if (0x4c == key) { // l p2 right
            a_dy = 1; a_ddy = MAX_MOVEMENT;
        }
    }
}

void write_bullet(int b, int x1, int y1, int z1, int x2, int y2, int z2, int color) {
    write_rect(b + 2, x1, y1, z1, ClosestValid(x2), ClosestValid(y2), ClosestValid(z2),
color);
}

const int BULLET_W = 40;
int flash = 0;
void FlashBullets() {
    int b;
    for (b = 0; b < 2; b ++) {
        switch (flash) {
            case 0:
                write_bullet(b, recX1[2 + b], recY1[2 + b], recZ1[2 + b], recX1[2 + b],
recY1[2 + b] + BULLET_W, recZ1[2 + b] + BULLET_W, raw_rgb_to_color(3, 0, 3));
                break;
            case 1:
                write_bullet(b, recX1[2 + b], recY1[2 + b], recZ1[2 + b], recX1[2 + b]
+ BULLET_W, recY1[2 + b], recZ1[2 + b] + BULLET_W, raw_rgb_to_color(0, 0, 3));
                break;
            case 2:
                write_bullet(b, recX1[2 + b] + BULLET_W, recY1[2 + b], recZ1[2 + b],
recX1[2 + b] + BULLET_W, recY1[2 + b] + BULLET_W, recZ1[2 + b] + BULLET_W,
raw_rgb_to_color(3, 2, 0));
                break;
            case 3:
```

```
                write_bullet(b, recX1[2 + b], recY1[2 + b] + BULLET_W, recZ1[2 + b],
recX1[2 + b] + BULLET_W, recY1[2 + b] + BULLET_W, recZ1[2 + b] + BULLET_W,
raw_rgb_to_color(0, 3, 3));
                break;
        }
    }
    flash = (flash + 1) % 4;
}

int PlayerHit() {
    int b;
    for (b = 0; b < 3; b ++) {
        if (0 == recZ1[b + 2]) {
            if ((recX1[0] < recX1[2 + b] && recX2[0] > recX1[2 + b]) || (recX1[0] <
recX1[2 + b] + BULLET_W && recX2[0] > recX1[2 + b] + BULLET_W)) {
                if ((recY1[0] < recY1[2 + b] && recY2[0] > recY1[2 + b]) || (recY1[0] <
recY1[2 + b] + BULLET_W && recY2[0] > recY1[2 + b] + BULLET_W)) {
                    return 1;
                }
            }
        }
    }
    return 0;
}

void HotPotato() {
    int i;
    for (i = 0; i < 4; i ++) {
        write_rect(i, 0, 0, 0, 0, 0, 0, rgb_to_color(0, 0, 0));
    }
    write_rect(4, 0, 0, 0, 254, 254, 0, rgb_to_color(3, 3, 0));
    write_rect(5, 254, 0, 0, 254, 254, 254, rgb_to_color(0, 3, 0));
    int iteration = 0, waitForHit = 0;
    while (HOT_POTATO == simulation) {
        HandleKey();
        if (0 == iteration % 50) {
            MoveHotPotatoShip();
            MoveAimer();
            RenderShipAim();
        }
        if (0 == iteration % 25) {
            MoveBullets();
        }
        if (0 == iteration % 10) {
            FlashBullets();
        }
        iteration = (iteration + 1) % 1000;
    }
}

void Reset() {
    int i;
    for (i = 0; i < 84; ++i) {
        IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, i, 255);
    }
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_X, ClosestValid(0));
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Y, ClosestValid(128));
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Z, ClosestValid(10));
}

int main() {
    init_keyboard();
    printf("Program running.\n");
```

```
    int done = 0;
    while (0 == done) {
        Reset();
        switch (simulation) {
            case 0:
                HotPotato();
                break;
            case 1:
                BouncingSquare();
                break;
            case 2:
                FlashingColumns();
                break;
            case 3:
                Driving();
                break;
            case 4:
                done = 1;
                break;
            case 5:
                DontCrash();
                break;
            default:
                HotPotato();
                break;
        }
    }
    return 0;
}
```

----

dontcrash.c - The demo game

```
#include "simulations.h"
#include "raytracer.h"
#include "altera_ps2/ps2_keyboard.h"
#include <stdio.h>

// x - depth, y - left, z - top

const int cam_y = 128;

void Delay() {
    int i;
    for (i = 0; i < 3000000; ++i) {}
}

const int car_height = 5;
const int car_length = 8;
const int car_width = 5;
const int car_depth = 10;
int car_color;

void DrawCar(int car_left) {
    // car back
    write_rect(1,
              car_depth, car_left, 0,
              car_depth, car_left + car_width, car_height,
              car_color);
    // car top
    write_rect(2,
```

```
                car_depth, car_left, car_height,
                car_depth + car_length, car_left + car_width, car_height,
                car_color);
    // car side
    if (car_left < cam_y) {
        write_rect(3,
                   car_depth, car_left + car_width, 0,
                   car_depth + car_length, car_left + car_width, car_height,
                   car_color);
    } else {
        write_rect(3,
                   car_depth, car_left, 0,
                   car_depth + car_length, car_left, car_height,
                   car_color);
    }
}

/**
 * Returns true (1) if there's any overlap between the range of numbers from a1
 * to a2 and from b1 to b2. Eg: Overlap(1, 3, 2, 4) -> true and
 * Overlap(1, 3, 4, 6) -> false.
 **/
int Overlap(a1, a2, b1, b2) {
    if (a1 < b1) {
        if (a2 > b1) {
            return 1;
        } else {
            return 0;
        }
    } else {
        if (a1 < b2) {
            return 1;
        } else {
            return 0;
        }
    }
}

void DontCrash() {
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Y, cam_y);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, CAM_Z, 10);
    car_color = rgb_to_color(0, 0, 3);
    write_rect(0, 0, 0, 0, 254, 254, 0, rgb_to_color(1, 1, 0)); // floor
    int car_left = 0; // This value only here to prevent warning
    int obstacle_left = 0; // This value only here to prevent warning
    int obstacle_depth = 0; // This value only here to prevent warning
    alt_u8 key;
    int y_overlap, x_overlap;
    int alive = 0;
    printf("Welcome to Don't Crash!\n");
    printf("Use A and D to move the car side to side.\n");
    printf("Don't crash into the obstacles!\n");
    printf("Press space to get started.\n");
    while (DONT_CRASH == simulation) {
        // Get input
        key = LatestKey();
        if (!alive) {
            if (key == KB_SPACE) {
                alive = 1;
                car_left = 125;
                obstacle_left = 110;
                obstacle_depth = 245;
            }
            continue;
```

```
        }
        // Update world
        switch (key) {
            case KB_A_DOWN:
                car_left--;
                break;
            case KB_D_DOWN:
                car_left++;
                break;
        }
        obstacle_depth--;
        if (obstacle_depth <= 0) {
            obstacle_depth = 230;
            obstacle_left = car_left;
        }
        // Draw to screen
        DrawCar(car_left);
        write_rect(4, obstacle_depth, obstacle_left, 0,
                   obstacle_depth, obstacle_left + 5, 10, rgb_to_color(3, 3, 3));
        // Check for end game
        x_overlap = Overlap(obstacle_depth, obstacle_depth,
                            car_depth, car_depth + car_length);
        y_overlap = Overlap(obstacle_left, obstacle_left + 5,
                            car_left, car_left + car_width);
        if (x_overlap && y_overlap) {
            printf("You crashed! Game over.\n");
            printf("Press space to restart or a number to switch to another game.\n");
            alive = 0;
        }
        // Call scheduler so other processes can run
        Delay();
    }
}
```

---

raytracer.h

```
#ifndef _RAYTRACER_H_
#define _RAYTRACER_H_
#include <system.h>
#include <io.h>
#define CAM_X 0
#define CAM_Y 2
#define CAM_Z 4
// r, g, and b must be at least 0 and at most 3
static alt_u8 raw_rgb_to_color(alt_u8 r, alt_u8 g, alt_u8 b) {
    return (r << 4 | g << 2 | b);
}
// The compiler should optimize away this function.
static void write_rect(unsigned rect_num,
                       alt_u8 x1, alt_u8 y1, alt_u8 z1,
                       alt_u8 x2, alt_u8 y2, alt_u8 z2,
                       alt_u8 color) {
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14, x1);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 2, y1);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 4, z1);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 6, x2);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 8, y2);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 10, z2);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 12, color);
}
```

```
// This craziness is to provide compile time range checking on the arguments to
// rgb_to_color.
// This can only take constants. It seems to work without compiler optimizations
// being turned on, but I don't know how.
// It works as follows:
// If one of the args is an illegal value, the expression will want to use the
// variable assertion_failed. This variable is listed as extern, but we don't
// actually define it anywhere. So we'll get a linker error.
// If all the values are ok, the expression will use the part which does the
// bit manipulation. In this case, the compiler never looks at the alternative
// and the linker never sees assertion_fails.
// Based on stuff from:
// http://www.embedded.com/columns/programmingpointers/164900888?_requestid=1002553
#define rgb_to_color(r, g, b) \
    (((r) >= 0 && (r) <= 3 && (g) >= 0 && (g) <= 3 && (b) >= 0 && (b) <= 3) ? \
     raw_rgb_to_color((r), (g), (b)) : \
     assertion_failed)
extern int assertion_failed;
#endif
```

---

simulations.h

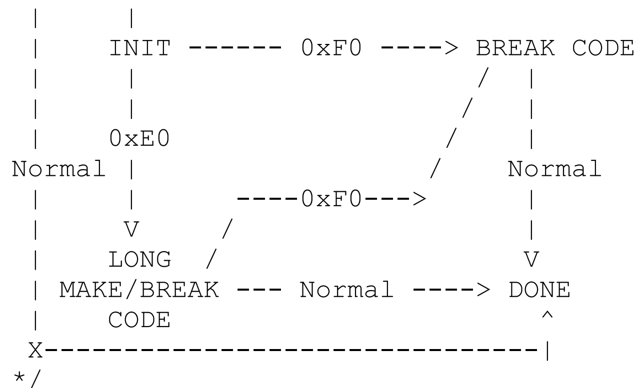
```
#ifndef SIMULATIONS_H_

#define SIMULATIONS_H_
#include <io.h>
extern int simulation;
#define HOT_POTATO 0
#define BOUNCING_SQUARE 1
#define FLASHING_CUBES 2
#define DRIVING 3
#define DONE 4
#define DONT_CRASH 5
extern alt_u8 LatestKey();
#endif /*SIMULATIONS_H_*/
```

---

ps2_keyboard.c

```
#include "ps2_keyboard.h"
#include <stdio.h>
#define SCAN_CODE_NUM  102
//////////////////////////////////////////////////////////////////////
// Table of scan code, make code and their corresponding values
// These data are useful for developing more features for the keyboard
//
alt_u8 *key_table[SCAN_CODE_NUM] = { "A", "B", "C", "D", "E", "F", "G", "H", "I", "J",
"K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V",
"W", "X", "Y", "Z", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "`", "-", "=",
"\\", "BKSP", "SPACE",
 "TAB", "CAPS", "L SHFT", "L CTRL", "L GUI", "L ALT", "R SHFT", "R CTRL", "R GUI", "R
ALT", "APPS", "ENTER",
"ESC", "F1", "F2", "F3", "F4", "F5", "F6", "F7", "F8", "F9", "F10", "F11", "F12",
"SCROLL", "[", "INSERT",
 "HOME", "PG UP", "DELETE", "END", "PG DN", "U ARROW", "L ARROW", "D ARROW", "R
ARROW", "NUM", "KP /", "KP *",
 "KP -", "KP +", "KP ENTER", "KP .", "KP 0", "KP 1", "KP 2", "KP 3", "KP 4", "KP 5",
"KP 6", "KP 7", "KP 8",
"KP 9", "]", ";", "'", ",", ".", "/" };
alt_u8 ascii_codes[SCAN_CODE_NUM] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I',
```

```
'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
 'W', 'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '`', '-', '=',
0, 0x08, 0, 0x09, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0x0A, 0x1B, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, '[', 0, 0,
0, 0x7F, 0, 0, 0, 0, 0,
 0, 0, '/', '*', '-', '+', 0x0A, '.', '0', '1', '2', '3', '4', '5', '6', '7', '8',
'9', ']', ';', '\'', ',',
 '.', '/' };
alt_u8 single_byte_make_code[SCAN_CODE_NUM] = { 0x1C, 0x32, 0x21, 0x23, 0x24, 0x2B,
0x34, 0x33, 0x43, 0x3B, 0x42, 0x4B, 0x3A, 0x31, 0x44, 0x4D,
 0x15, 0x2D, 0x1B, 0x2C, 0x3C, 0x2A, 0x1D, 0x22, 0x35, 0x1A, 0x45, 0x16, 0x1E, 0x26,
0x25, 0x2E,
 0x36, 0x3D, 0x3E, 0x46, 0x0E, 0x4E, 0x55, 0x5D, 0x66, 0x29, 0x0D, 0x58, 0x12, 0x14,
0, 0x11,
0x59, 0, 0, 0, 0, 0x5A, 0x76, 0x05, 0x06, 0x04, 0x0C, 0x03, 0x0B, 0x83, 0x0A, 0x01,
0x09, 0x78,
 0x07, 0x7E, 0x54, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x77, 0, 0x7C, 0x7B, 0x79, 0, 0x71,
0x70, 0x69,
0x72, 0x7A, 0x6B, 0x73, 0x74, 0x6C, 0x75, 0x7D, 0x5B, 0x4C, 0x52, 0x41, 0x49, 0x4A };
alt_u8 multi_byte_make_code[SCAN_CODE_NUM] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0x1F, 0, 0, 0x14, 0x27, 0x11, 0x2F, 0, 0, 0,
0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0x70, 0x6C, 0x7D, 0x71, 0x69, 0x7A, 0x75, 0x6B, 0x72, 0x74,
0, 0x4A, 0, 0,
 0, 0x5A, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
////////////////////////////////////////////////////////////////////
// States for the Keyboard Decode FSM
typedef enum
{
        STATE_INIT,
        STATE_LONG_BINARY_MAKE_CODE,
        STATE_BREAK_CODE ,
        STATE_DONE
} DECODE_STATE;
// Maintain KB decode FSM state globally for non-blocking kb read
DECODE_STATE state = STATE_INIT;
//helper function for get_next_state
alt_u8 get_multi_byte_make_code_index(alt_u8 code)
{
        alt_u8 i;
        for (i = 0; i < SCAN_CODE_NUM; i++ )
        {
                if ( multi_byte_make_code[i] == code )
                        return i;
        }
        return SCAN_CODE_NUM;
}
//helper function for get_next_state
alt_u8 get_single_byte_make_code_index(alt_u8 code)
{
        alt_u8 i;
        for (i = 0; i < SCAN_CODE_NUM; i++ )
        {
                if ( single_byte_make_code[i] == code )
                        return i;
        }
        return SCAN_CODE_NUM;
}
//helper function for read_make_code
/* FSM Diagram (Main transitions)
 * Normal bytes: bytes that are not 0xF0 or 0xE0
  ______
  |     |
```



```
  |     |
  |    INIT ------ 0xF0 ----> BREAK CODE
  |     |                    /   |
  |     |                   /    |
  |    0xE0                /     |
Normal |                  /    Normal
  |     |      ----0xF0--->      |
  |     V     /                  |
  |    LONG  /                   V
  | MAKE/BREAK --- Normal ----> DONE
  |    CODE                       ^
  X-------------------------------|
 */
DECODE_STATE get_next_state(DECODE_STATE state, alt_u8 byte, KB_CODE_TYPE
*decode_mode, alt_u8 *buf)
{
        DECODE_STATE next_state = STATE_INIT;
        alt_u16 idx = SCAN_CODE_NUM;
        switch (state)
        {
                case STATE_INIT:
                        if ( byte == 0xE0 )
                        {
                                next_state = STATE_LONG_BINARY_MAKE_CODE;
                        }
                        else if (byte == 0xF0)
                        {
                                next_state = STATE_BREAK_CODE;
                        }
                        else
                        {
                                idx = get_single_byte_make_code_index(byte);
                                if ( (idx < 40 || idx == 68 || idx > 79) && ( idx !=
SCAN_CODE_NUM ) )
                                {
                                        *decode_mode = KB_ASCII_MAKE_CODE;
                                        *buf= ascii_codes[idx];
                                }
                                else
                                {
                                        *decode_mode = KB_BINARY_MAKE_CODE;
                                        *buf = byte;
                                }
                                next_state = STATE_DONE;
                        }
                        break;
                case STATE_LONG_BINARY_MAKE_CODE:
                        if ( byte != 0xF0 && byte!= 0xE0)
                        {
                                *decode_mode = KB_LONG_BINARY_MAKE_CODE;
                                *buf = byte;
                                next_state = STATE_DONE;
                        }
                        else
                        {
                                next_state = STATE_BREAK_CODE;
                        }
                        break;
                case STATE_BREAK_CODE:
                        if ( byte != 0xF0 && byte != 0xE0)
                        {
                                *decode_mode = KB_BREAK_CODE;
                                *buf = byte;
                                next_state = STATE_DONE;
```

```
                        }
                        else
                        {
                                next_state = STATE_BREAK_CODE;
                        }
                        break;
                default:
                        *decode_mode = KB_INVALID_CODE;
                        next_state = STATE_INIT;
        }
        return next_state;
}
int read_make_code(KB_CODE_TYPE *decode_mode, alt_u8 *buf)
{
        alt_u8 byte = 0;
        int status_read =0;
        *decode_mode = KB_INVALID_CODE;
        DECODE_STATE state = STATE_INIT;
        do
        {
                status_read = read_data_byte_with_timeout(&byte, 0);
                //FIXME: When the user press the keyboard extremely fast, data may get
                //occasionally get lost
                if (status_read == PS2_ERROR)
                        return PS2_ERROR;
                state = get_next_state(state, byte, decode_mode, buf);
        } while (state != STATE_DONE);
        return PS2_SUCCESS;
}
int read_make_code_no_block(KB_CODE_TYPE *decode_mode, alt_u8 *buf) {
    alt_u32 data_reg = 0;
    alt_u16 num = 0;
    *decode_mode = KB_INVALID_CODE;
    alt_u8 byte = 0;
    data_reg = read_data_reg();
    num = read_num_bytes_available(data_reg);
    while (num > 0) {
        byte = read_data_byte(data_reg);
//        printf("In state %d. Processing byte: %x\n", state, byte);
        state = get_next_state(state, byte, decode_mode, buf);
        if (state == STATE_DONE) {
            state = STATE_INIT;
            return PS2_SUCCESS;
        }
        num--;
    }
    return PS2_NOT_READY;
}
alt_u32 set_keyboard_rate(alt_u8 rate)
{
//      alt_u8 byte;
        // send the set keyboard rate command
        int status_send = write_data_byte_with_ack(0xF3, DEFAULT_PS2_TIMEOUT_VAL);
        if ( status_send == PS2_SUCCESS )
        {
                // we received ACK, so send out the desired rate now
                status_send = write_data_byte_with_ack(rate & 0x1F,
DEFAULT_PS2_TIMEOUT_VAL);
        }
        return status_send;
}
alt_u32 reset_keyboard()
{
        alt_u8 byte;
```

```
        // send out the reset command
        int status = write_data_byte_with_ack(0xff, DEFAULT_PS2_TIMEOUT_VAL);
        if ( status == PS2_SUCCESS)
        {
                // received the ACK for reset, now check the BAT result
                status = read_data_byte_with_timeout(&byte, DEFAULT_PS2_TIMEOUT_VAL);
                if (status == PS2_SUCCESS && byte == 0xAA)
                {
                        // BAT succeed
                }
                else
                {
                        // BAT failed
                        status = PS2_ERROR;
                }
        }
        return status;
}
```

---

ray_tracer.h

```
#ifndef _RAYTRACER_H_
#define _RAYTRACER_H_
#include <system.h>
#include <io.h>
#define CAM_X 0
#define CAM_Y 2
#define CAM_Z 4
// r, g, and b must be at least 0 and at most 3
static alt_u8 raw_rgb_to_color(alt_u8 r, alt_u8 g, alt_u8 b) {
    return (r << 4 | g << 2 | b);
}
// The compiler should optimize away this function.
static void write_rect(unsigned rect_num,
                       alt_u8 x1, alt_u8 y1, alt_u8 z1,
                       alt_u8 x2, alt_u8 y2, alt_u8 z2,
                       alt_u8 color) {
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14, x1);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 2, y1);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 4, z1);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 6, x2);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 8, y2);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 10, z2);
    IOWR_16DIRECT(RAY_TRACER_AVALON_MODULE_INST_BASE, 6 + rect_num * 14 + 12, color);
}
// This craziness is to provide compile time range checking on the arguments to
// rgb_to_color.
// This can only take constants. It seems to work without compiler optimizations
// being turned on, but I don't know how.
// It works as follows:
// If one of the args is an illegal value, the expression will want to use the
// variable assertion_failed. This variable is listed as extern, but we don't
// actually define it anywhere. So we'll get a linker error.
// If all the values are ok, the expression will use the part which does the
// bit manipulation. In this case, the compiler never looks at the alternative
// and the linker never sees assertion_fails.
// Based on stuff from:
// http://www.embedded.com/columns/programmingpointers/164900888?_requestid=1002553
#define rgb_to_color(r, g, b) \
    (((r) >= 0 && (r) <= 3 && (g) >= 0 && (g) <= 3 && (b) >= 0 && (b) <= 3) ? \
```

```
     raw_rgb_to_color((r), (g), (b)) : \
     assertion_failed)
extern int assertion_failed;

#endif
```