CUDoom: A Raycasting video game

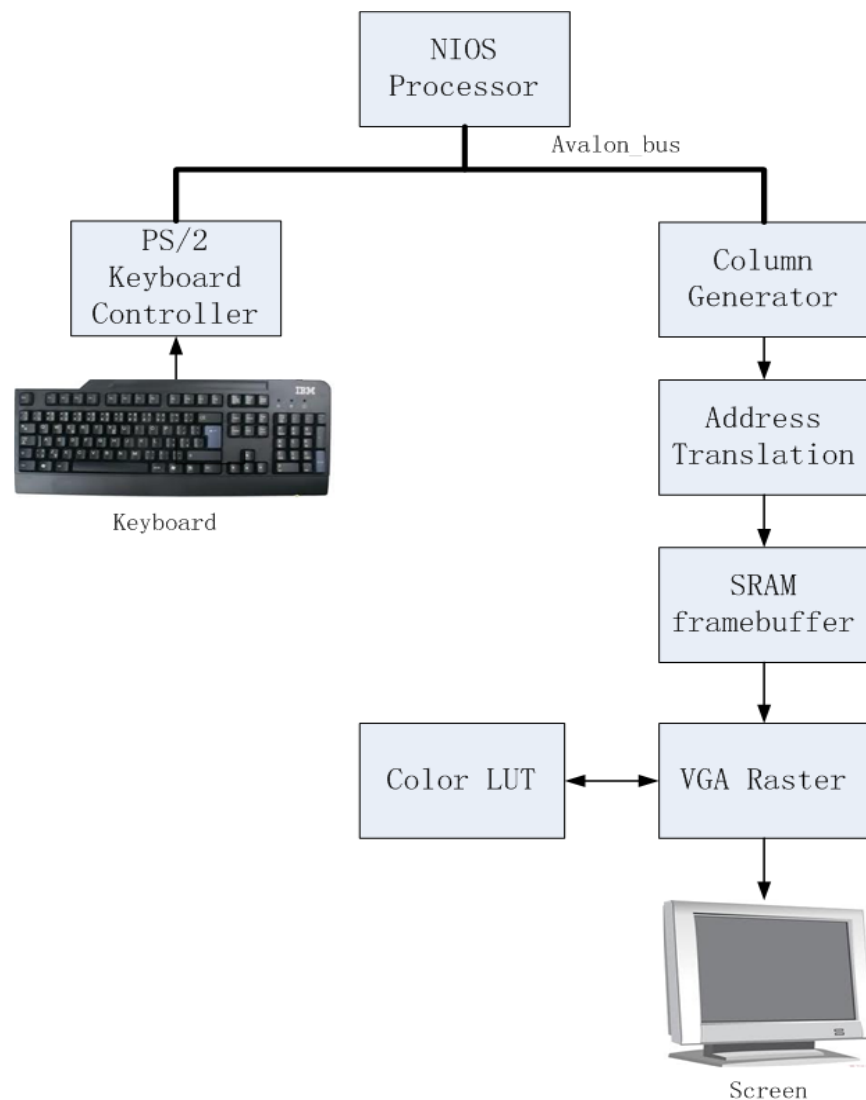
CSEE 4840 Embedded System Design
3/26/12
Project Design

Alden Goldstein (ag3287)
Edward Garcia (ewg2115)
Minyun Gu (mg3295)
Wei-Hao Yuan (wy2211)
Yiming Xu (yx2213)

I. High-Level Overview

Our project will implement a video game using ray casting techniques. The game will accept keyboard input from users and display a 3D like world on a VGA output. Raycasting creates a pseudo 3D world from a 2D map. At the expense of fixing the camera rotation around the player's vertical axis, computation is greatly decreased while maintaining the illusion of a 3D world. In the implementation of this project, all the walls will have the same height and will be orthogonal squares on a 2D grid. Keyboard input will be handled by hardware as well.

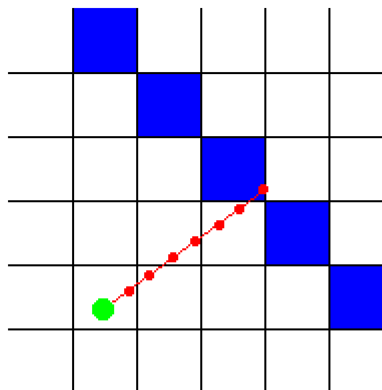
The following is the block structure of our system:



The system mainly includes three parts: NIOS processor and its framebuffer in RAM, as well as two peripherals VGA output and keyboard input. Users will interact with the system by keyboard inputs. Once any arrow key is pressed, the corresponding signals will be passed to processor via interrupts sent by the keyboard controller. The processor will deal with the action to move player's position correspondingly. It will then re-calculate distances to walls based on the raycasting algorithm. The RGB data for all pixels in the new frame can be updated to column by column after processed in column generator. Afterwards, address translation writes those data to the proper locations of framebuffer. Finally, the VGA raster will first look up respective physical color of each pixel from color Look-up Table (LUT) according to the corresponding color data fetched from the framebuffer. The VGA raster will continuously display updated frames from the processor..

II. Software

We will use ray extension as a way of determining where the walls are. This is a way of incrementing the rays in a direction to find an approximate distance to the wall. The larger the ray increments are, the more inexact the measurements will be, and the more likely part of a wall will be missed. However smaller increments will be a lot slower. We found that extending the rays at 1/16 of the square side unit on the map is about the largest increment that we can use that still gives a very nice image.



This algorithm will be the bulk of computation in C. Outside of the ray extension loop consists of constant time operations involving, reads, writes, integer arithmetic, and only 2 divisions. Using a Nios II/f processor which includes a data cache, most of these should be one cycle operations excluding the division. After looking successfully writing the fixed-point code in software, we can now estimate this to be around 120 clock cycles,

$$2\ divisions * (20\ cycles/division) + 80\ single\ cycle\ operations = 120\ clock\ cycles$$

Now let's look at the loop. The loop overhead constitutes of two (barrel) shifts, two multiplications to retrieve the address of the two dimensional array, a single comparison statement, and a single read (from data cache).

```
while (worldMap[rayPosX>>precisionShift][rayPosY>>precisionShift]==0)
{
       count += count_step;
       rayPosX += rayDirX;
       rayPosY += rayDirY;
}
```

$Loop\ Overhead\ =\ 2\ barrel\ shifts\ +\ 2\ multiplications\ +1\ read\ +\ 1\ comparison\ +2\ cycles\ for\ while\ statement(?)$
$Loop\ Overhead\ \approx 8\ cycles$

The instructions inside the loop involve three additions. If we assume the “count_step”, “rayDirX”, and “rayDirY” are stored in registers, there may not be overhead for memory reads. However we will be conservative and assume there are read operations.

$Loop\ Execution\ =\ 3\ additions\ +\ 3\ reads\ =\ 6\ cycles$
$Total\ Time\ for\ Single\ Iteration\ \approx\ 14\ cycles$

Now assume we have the worst case (a large room, looking straight ahead), and on average we have a ray extension of say 13 side units (this would roughly be the average ray length for a 10 x 10 room).

$$Total\ Loop\ Time\ (Worst\ Case)\ \approx\ 14\ cycles\ \times 16\ iterations/side\ unit\ \times 13\ side\ units\ \approx 2912\ cycles$$

$$Total\ Compute\ Time\ for\ Column\ = 2912\ cycles\ for\ Ray\ Extension\ +\ 120\ cycles\ for\ other\ operations$$

$$Total\ Compute\ Time\ for\ Column = 3032\ cycles$$

Now we have to consider all columns being computed.

$$Total\ Compute\ Time\ for\ Software = 3032\ cycles/column\ \times 640\ columns\ = 1,940,480\ cycles$$

Now we can divide this number into the clock rate to get the software limit for the frame rate. Notice that we do not need to add the clock cycles to the hardware calculation. This is because the hardware will be doing calculations independently. Hardware will wait for column information from software. And once received, the hardware will perform calculations and output to the frame buffer while the software computes the next column. Thus, the limiting factor for each column will be the slowest computations between hardware and software.

$$Limited\ Frame\ Time\ for\ Software\ =\ 50\ MHz/1,940,480\ cycles\ \approx 26\ frames\ per\ second$$

This isn’t that bad for being the worst case! If we are right, then we should not worry about frame computation in software for now.

III. Timing (Critical Path Discussion)

In our project we aim to provide a framerate of 30 Frames Per Second (FPS). Above we showed that at worst case, the software should perform at around this level. For hardware, it is important to consider the number of clock cycles per pixel, because we can dictate this value directly. With this number fixed, we can compute the upper bounds of clock cycles to find the color of an individual pixel. In an implementation where the raycasting algorithm is not used and the color of each pixel is computed individually, the following equation can be solved to find the maximum number of clock cycles that can be spent to compute the color a pixel while maintaining a framerate of 30 FPS.

$$1\ Frame\ /\ 307200\ Pixels * \ 1\ Pixel/\ X\ Clock\ Cycles\ * 50,000,000\ Clock\ Cycles/\ 1\ second\ =\ 30\ Frames/1\ Second$$

Solving the equation yields X = 5.42 Clock Cycles/Pixel. In a raycasting implementation, the color values of an entire column is computed at a time. The following equation can be solved to find the maximum number of clock cycles that can be spent to compute the color values of an entire column while maintaining a framerate of 30 FPS.

$$1\ Frame\ /\ 640\ Columns * \ 1\ Column/\ X\ Clock\ Cycles\ * 50,000,000\ Clock\ Cycles/\ 1\ second\ =\ 30\ Frames/1\ Second$$

Solving the equation yields X = 2604.17 Clock Cycles/Column. This number represents a target value to keep in mind while writing the code to complete the raycasting engine.

IV. Memory

As of now, our C code is small enough to have the processor use block ram. We have four sinusoid lookup tables currently ($4 \times 3480\ angles\ \times 4\ bytes\ each$) . We use full period sine and cosine tables, but we can compress this to a quarter period of just a sine table and just keep track of angle shifts and signs.

$$total\ for\ sinusoids\ = 2 \times 3480\ angles/4\ \times 4\ bytes\ each\ =\ 6960\ bytes\ = 55,680\ bits$$

Now we have to consider a worldmap, which can be stored in a 24 X 24 grid of single bytes.

This overhead would be...

$$grid\ overhead\ =\ 24\ \times 24 \times 8\ bits/byte\ = 4608\ bits$$

Together this is around 60,000 bits, the additional variables are negligible. The DE2 board has about 400,000 bits of block ram, so this leaves us room to do plenty of other things in block ram.

512 KB SRAM

16 bits Wide

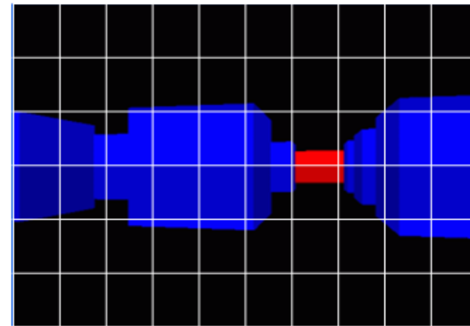
256 K Rows

8 bit Color Value	8 bit Color Value
8 bit Color Value	8 bit Color Value
8 bit Color Value	8 bit Color Value
8 bit Color Value	8 bit Color Value
8 bit Color Value	8 bit Color Value

SRAM Framebuffer

640*480 = 307,200 Pixels

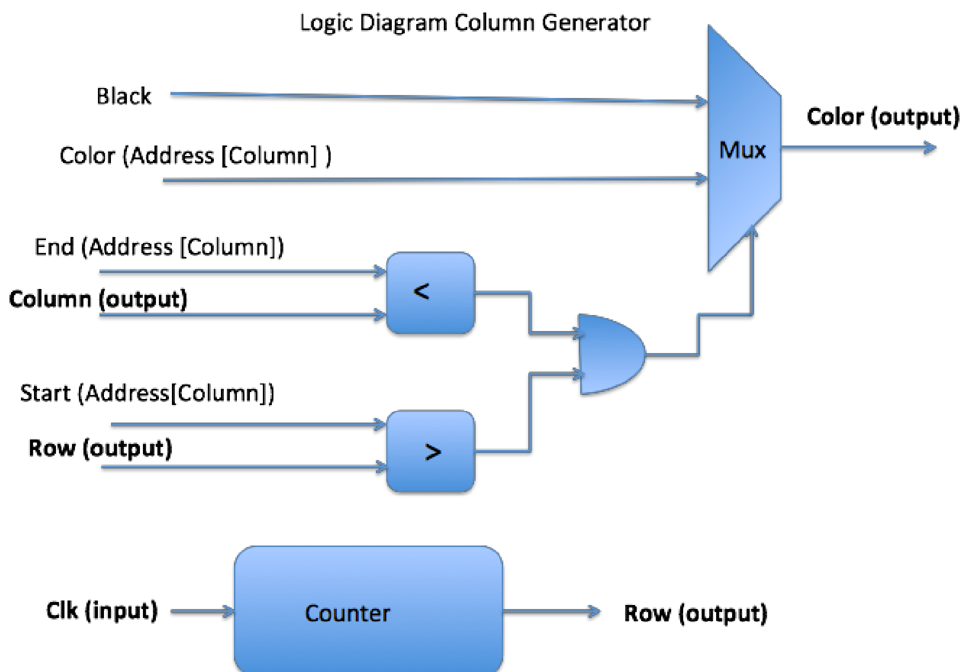
640 Pixels

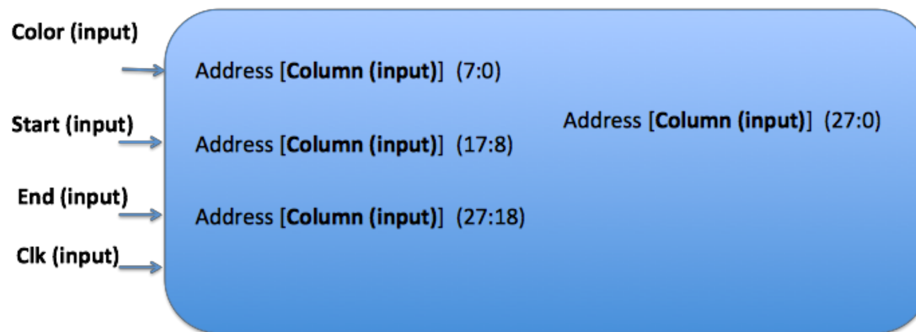


480 Pixels

One of the main components in the design will be the framebuffer, which will be used to hold the color values for each pixel on the screen. Using a screen resolution of 640 by 480 bytes, there is a total of 307,200 pixels that need to be displayed. The SRAM provided on the DE2 board was chosen to hold this buffer since it was large enough to hold this buffer and provides an access rate of 25 MHz which is similar to the VGA timing. The SRAM is divided into 256K rows which each have 16 bits each. This gives a total of 512KB of storage space on the SRAM. This limits the color value to represent a single pixel to 8 bits. Since the VGA raster block requires 10 bit RGB values, a lookup table will be used to convert 8 bit color values into the required 10 bit RGB values.

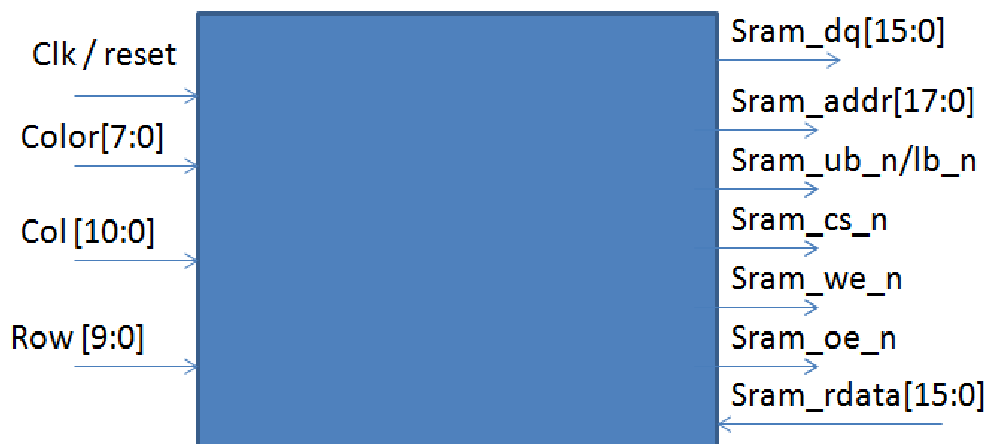
V. Column Generator





The column generator translates the start position and end position of the wall to be rastered and populates the framebuffer column by column. By using this component we are able to minimize the number of writes that nios processor makes to the framebuffer.

VI. Address Generation



The address translation transmit position information and color information to SRAM buffer. The VGA raster would read data from the SRAM and recalculate it with LUT and generate 10 bit RGB data for VGA. This would allow for storage saving in SRAM, while being compatible with the RGB data for the VGA.

VII. System I/O

This system will generate a data token containing the following information: the Start point of a column, End point of a column, Color information of a column and Column number. All of information will be stored in a buffer in the Column Generator that hardware will access to populate the SRAM framebuffer. Since the buffer is small, we will generate it with block ram.

bit[39:38]	bit[37:28] **(address)**	bit[27:18]	bit[17:8]	bit[7:0]
N/A	Column #	End point of a column	Start point of a column	Color Information