6.111 Project Proposal
Realistic 3D Gaming

Daniel Whitlow
Ranbel Sun

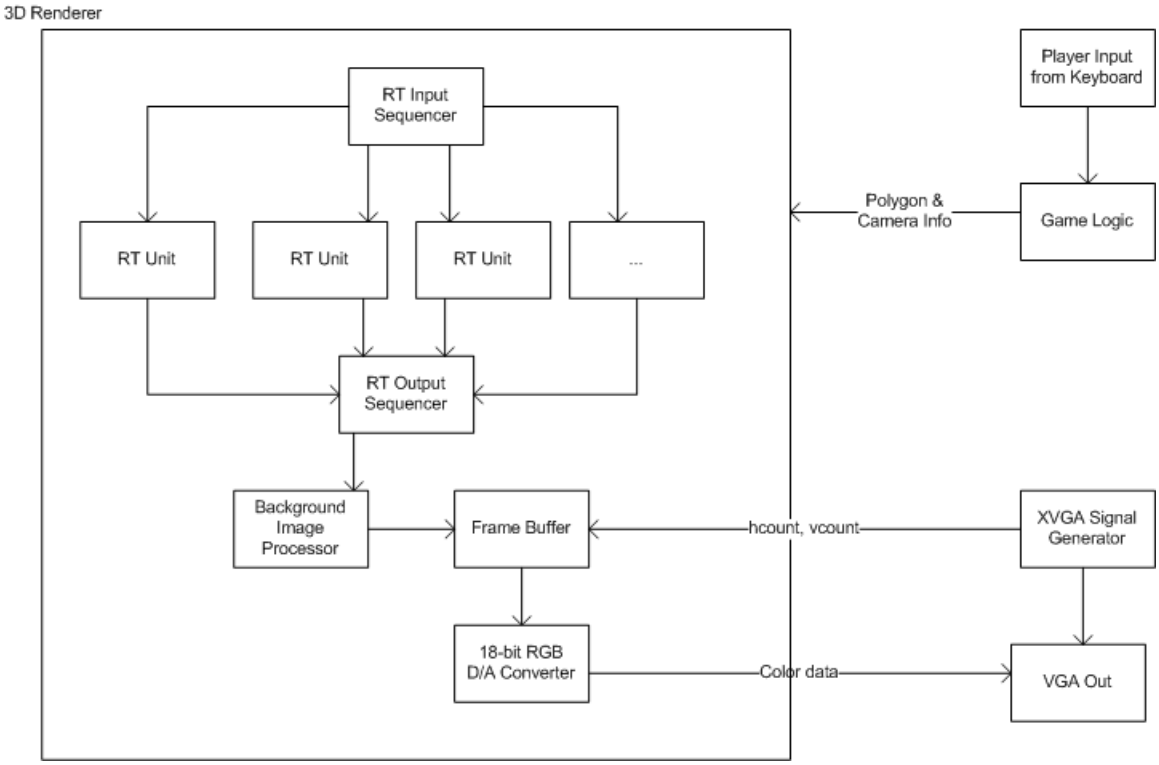
Overview

The goal of our project is to create a realistic 3-D game using ray-tracing as the rendering technique. Although ray-tracing is generally more computationally intensive than rasterization, it can produce more realistic reflections and lighting effects and can be parallelized very well. To take advantage of this, our design will have multiple ray-tracing modules computing correct pixel values in parallel, with sequencers controlling the inputs and outputs to and from the ray-tracers. Since there is a trade-off between visual realism and frame rate, we will need to find the optimum way to generate 3-D graphics while still ensuring game playability. We will first implement basic shading and reflectivity effects for geometric shapes, and if time and resources permit, add additional visuals such as texture. The game itself will be based off of Starfox by Nintendo, in which the player controls a plane flying into the screen and dodges obstacles in its path. A PS/2 keyboard will be used as player input, and the visuals will be shown on a 1024x768 VGA monitor.

Description

The project will be split up into two main parts: the game logic, and the 3D renderer. As indicated in our block diagram below, we believe the renderer will take the most time. We will use 18-bit color to align with the ZBT memory's dimensions. The code for the XVGA signal generator will be taken from Lab 5, and we will also use the PS/2 keyboard module provided from an earlier 6.111 term. Thus, the modules we will actually be coding are the Game Logic, Ray Tracer Input Sequencer, Ray Tracer Units, Ray Tracer Output Sequencer, Background Image Processor, Frame Buffer, and 18-bit RGB D/A Converter.

Block Diagram



Ray Tracer Input Sequencer

Since ray tracing can be a very time consuming process, we intend to parallelize it by using as many ray tracing modules as we can fit onto the FPGA. The input sequencer controls all of these units, assigning a pixel to calculate to the first non-busy unit, then determining the next pixel that needs to be calculated and repeating. To do this, it maintains an x and y counter. When a row of pixels has been calculated, it moves down to the next one. When all pixels have been calculated, the input sequencer waits until the frame buffer switches frames, then resets. This module should be fairly simple and use no memory outside of a few registers. Ranbel will be responsible for this module.

Ray Tracer Unit

This unit will accept a pixel [an (x, y) coordinate] to calculate the color of, and use a ray tracing algorithm to determine what the color should be. It also takes camera, polygon, and light source info directly from the game logic. Its outputs are a busy and done signal, along with a color, the x, y coordinates the color was calculated for, and a bit to indicate whether the calculated ray intersected anything. Due to the number of complex arithmetic operations this module is expected to perform, we expect this will be one of the most complex units in our project, in terms of both chip space and implementation time. To save space, we will attempt to use a pipelined multiplier and divider and use the same set of these for all multiplications and divisions. Instead of that, we may attempt to pipeline

the entire ray tracer unit in order to fully utilize each one. Daniel will be responsible for this module.

Ray Tracer Output Sequencer

Since the memory will probably not have ports to allow each ray tracer unit to write its output to it immediately, this module will buffer the output of each ray tracer unit and write the results to memory one at a time. We assume that the rate the ray tracers produce output collectively at a rate of less than 1 result per cycle, so we only need one buffer per ray trace unit. This means that this module will use 40 bits (the color bits, the intersection bit, and the coordinates, plus an additional bit to indicate that the data in the registers is valid) times the number of ray trace units of BRAM. The output sequencer will constantly cycle through its registers and output the data from the first valid register it encounters every clock cycle. Ranbel will be responsible for this module.

Background Image Processor

The background image processor sits between the output sequencer and the frame buffer, and is used to display a background in the pixels that are not displaying part of a 3d scene. If the intersection bit is 0, instead of writing black to memory, this module will dynamically generate a color based on the (x, y) coordinate provided. This unit will be designed to be simple (no extensive storage space, no high-latency operations). If it cannot be simplified for some reason, it will simply be removed. Ranbel will be responsible for this module.

Frame Buffer

The frame buffer is used to store the images that will be displayed on screen. It will store the data for two frames: the one which is currently being written, and the one which is currently being read. Each frame will be stored on a separate ZBT RAM. A status bit will be used to indicate which buffer is the read buffer (and by extension, which other buffer is the write buffer). When the write buffer finishes being written to (when the last pixel is written to memory), the status bit will be flipped and a signal will be sent to the ray tracer input sequencer. This module will use up both of the ZBT chips available. Ranbel will be responsible for this module.

18-bit RGB D/A Converter

This module takes an 18-bit RGB color code (6 bits for each color), and then generates an analog voltage for each color, and sends it to VGA out. We don’t expect it to take up too much space or memory, or be too complex to implement. Ranbel will be responsible for this module.

Game Logic

This module will be responsible for processing player input and translating it to camera, polygon, and light source data to output to the 3D renderer. We plan to represent all of the objects in the game as simple geometric shapes in order to avoid generating and storing data for models. We expect this to be similar in complexity to lab 5. Both of us will work on it.

External Components

We will be using a keyboard for player input—probably just the one that's plugged into the logic analyzer.

Testing

To test the ray tracing concepts, we will first create a simulator for a hardware ray tracer in Java. This will allow us to debug a lot of the concepts without having hardware complexities intervene. Once implemented in Verilog, the modules can be tested independently using ModelSim, given an appropriate set of inputs.