Balkentroll:
Lords of Acid

Technical Design

Piotr Mitros
Kristin Carr

With Original Musical Score by

Daniel Roy

May 15, 2003

Abstract

We implemented a very simple acid-themed 3D racing game. The game runs at 80's era 640×480 resolution, with 32bpp and 60fps. It is based on a crude 3D engine that does not support rotations (or camera movement beyond the x-y plane), and does no interpolation on texture mapping. It has basic shadows, very crude linear lighting, and even cruder transparency. Rudimentary support for LCD shutter glasses is also included. It is combined with an advanced sound system that modifies the audio based on the game play.

Contents

List of Figures

1 Acknowledgements

Since entering MIT, our verbal skills have fallen from on par with the top couple of percent of the US population to slightly below those of an annual Loebner Prize winner. As such, we decided to plagerize the acknowledgements section from the most excellent Olin Shivers[1]:

> Who should I thank? My so-called "colleagues," who laugh at me behind my back, all the while becoming famous on my work? My worthless graduate students, whose computer skills appear to be limited to downloading bitmaps off of netnews? My parents, who are still waiting for me to quit "fooling around with computers," go to med school, and become a radiologist? My department chairman, a manager who gives one new insight into and sympathy for disgruntled postal workers?
>
> My God, no one could blame me—no one!—if I went off the edge and just lost it completely one day. I couldn't get through the day as it is without the Prozac and Jack Daniels I keep on the shelf, behind my Tops-20 JSYS manuals. I start getting the shakes real bad around 10am, right before my advisor meetings. A 10 oz. Jack 'n Zac helps me get through the meetings without one of my students winding up with his severed head in a bowling-ball bag. They look at me funny; they think I twitch a lot. I'm not twitching. I'm controlling my impulse to snag my 9mm Sig-Sauer out from my day-pack and make a few strong points about the quality of undergraduate education in Amerika.
>
> If I thought anyone cared, if I thought anyone would even be reading this, I'd probably make an effort to keep up appearances until the last possible moment. But no one does, and no one will. So I can pretty much say exactly what I think.
>
> Oh yes, the acknowledgements. I think not. I did it. I did it all, by myself.

2 Overview

2.1 Game Play

The idea behind Bulk'n Troll is simple. There is a playing field with a ball on it. The player controls the ball, and can accelerate forwards, backwards, left, right or jump. The player cannot turn. If the player falls off of the playing field, the player starts over from the beginning.

As such, the game essentially follows the basic design of many previous games, such as SkyRoads and Babel.

The playing field consists of two levels. The bottom one is solid, and is texture-mapped with a gradient texture, while the upper one consists of energy fields, and therefore is semi-

[1]Idea for "borrowing" acknowledgements stolen from Philip Greenspun.

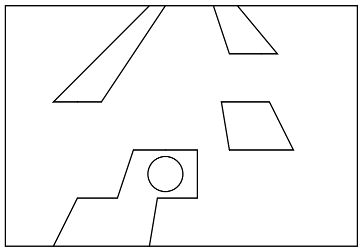

Figure 1: Mock Up of the Balken Troll Game

transparent. The player can jump up through the energy fields (which were primarily added to make the game easier to play). A portion of the bottom level snakes back and forth and is animated.

In addition, the player can hold down a boost button prior to jumping, which allows the player to jump higher. This was primarily added to prevent players from spending most of their time in the air.

2.2 Design Overview

2.2.1 Design Philosophy

We believe good engineering is characterized by simplicity. We optimized our code for providing as much functionality as possible, with minimal code complexity. We believe we achieved a reasonably large level of functionality for the simplicity of the code involved. The entire body of VHDL code (render engine, game play engine and sound system) fits in under 2000 lines of code.

We also implemented the code based on a bottom-up philosophy, with very little design up-front. We quickly hacked in features. Once each feature was implemented, we would go back, and clean up the code, based on our new understanding of what the feature entailed. We believe this led to clearer, shorter and simpler code.

2.3 Global Design

The overall design is shown in figure 2. The game play engine outputs x, y and z coordinates for the ball, as well as the velocities in all three axes. It outputs these to the rendering module and sound module. The rendering module generates the video signals, calculating the color of each pixel on-the-fly as it outputs to the monitor (this is done because we do not have sufficient memory or memory bandwidth for a usable frame buffer).

The sound module is stand-alone, while the rendering module sends a signal at the beginning of each refresh to the game play module, so that the game play can update the system state between refreshes. For reasons that will be explained later, it also signals the game play module as to whether the race tracks exist at the current x, y coordinates of the ball.

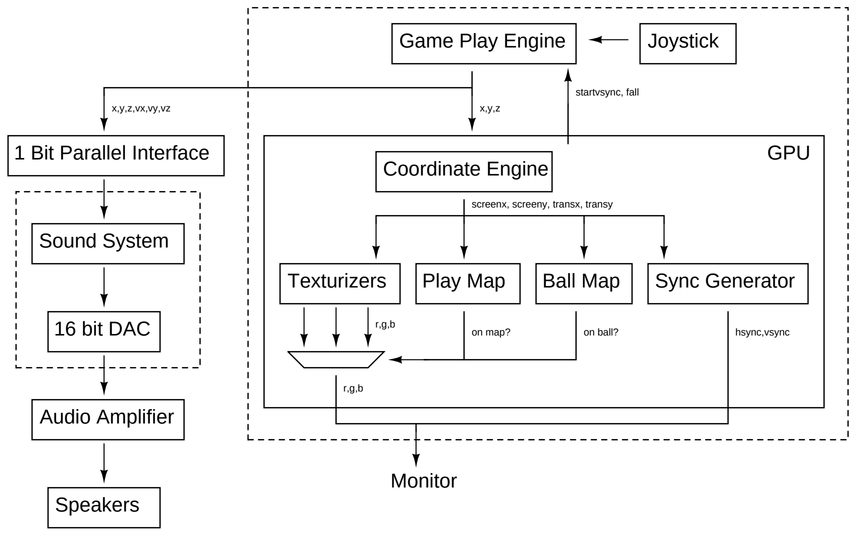

Figure 2: Overall Block Diagram

3 Pixel Pipeline

The rendering engine block diagram is shown in figure 3.

Note that the pipeline is uneven; some signals propagate through as many as three more stages than others. This is intentional. The render engine computes whether or not the ball is over each level of the playing field near the end of the pipeline. Unevenness in the pipeline results in pixels in different parts of the pipeline being shifted relative to each other in the X coordinate. This manifests itself as a slightly different starting position for the ball, and a slightly different race track alignment than if the pipeline were even. However, it does not cause errors in game play physics, since the game play physics are set by the output of the render engine, rather than by some model stored elsewhere in the system

Since each LE has an output register, pipelining is nearly free[2]. As such, we stuck a register after almost every operation[3].

3.1 Coordinate Translation

We wanted to translate the 2 dimensional screen coordinates into a 3D playing field, as shown in figure 4.

This is obviously impossible to do within the FPGA implemented näively as a raw transformation, due to size and space limitations. After spending some time doing math, we found that it was easy to do the transformation incrementally, from the previous pixel. If we walk across the screen from left to right on a single line, we find that for each unit in one coordinate step, we move by a fixed offset in the other. Going from 3D to 2D, this is simply d, the distance from the horizon. Going from 2D to 3D, this is $\frac{1}{d}$.

[2]Except for interconnect usage.

[3]This was impossible in some places, due to bugs in the Altera tools

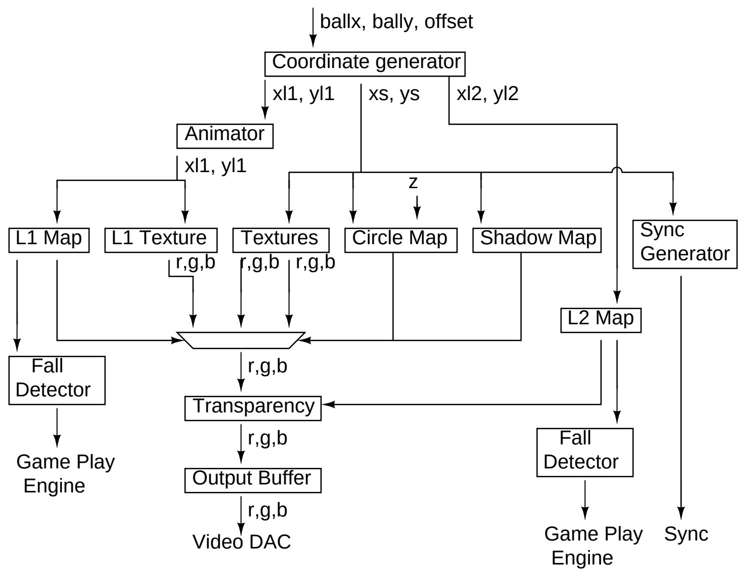

Figure 3: Rendering Pipeline

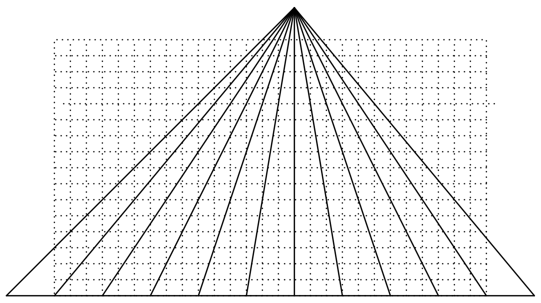

Figure 4: 3D Coordinates Superimposed Over 2D Coordinates

This was a nice result, but still gave us a division per line. Divisions in digital suck. They really suck. They are generally implemented using a large table, followed by successive approximations. This is slow, and very gate intensive. Since we only had one division per line, we could tolerate slow. However, we were not prepared to pay the price in gates, nor in implementation time. Fortunately, we noticed that since we only needed one coefficient per line, we could precompute the coefficients and store the result in ROM. We paid for this approach with:

- 480 words of ROM. This is a huge portion of memory, since the FPGA only has 12 kilobits.
- Camera angle was now partially fixed.

When we precomputed the result, we increased number of bits until there were no visible errors displaying a rectangular grid. For safety, we added one additional bit.

Conveniently, if we want the lines going into the event horizon to be straight, we find that, in any line:

$$\frac{dx_{3d}}{dx_{2d}} = \alpha \frac{dy_{3d}}{dy_{2d}}$$

For some constant α. We simply let $\alpha = 1$, and so reuse the coefficients from the table directly.

Furthermore, we find that since lines at other heights converge into the same vanishing point, the set of lines is the same (but with a different correspondence to screen coordinates). Working through the math, we find that we must have:

$$\frac{dx_{3d}}{dx_{2d}} = \beta \frac{dx'_{3d}}{dx'_{2d}}$$

For a constant β. As such, we can again reuse the table, scaling with $\beta = 3/4$ (following our rule of only allowing multiplications that can be expressed as a single addition, with bit shifts).

3.2 Animation

Animation is handled by adding an offset to the translated x coordinate if the y coordinate is within the appropriate range for the snaking portion of the course. To generate the offset, we use the sum of the y coordinate and the clock counter offset by 11 bits. We use the 19th bit of this to detect overflow, and if we overflow, we invert the entire vector. This causes the track to snake back and forth.

The snake looks curved due to rounding errors. We decided against increasing the number of bits, since the curved effect looked better than a simple angled track.

3.3 Race Track Map

The play maps are stored in a ROM on-board the FPGA. The modules pass several middle and higher order bits of the translated x and y coordinates to the ROMs. If the value of the x coordinate is within the range of the map, the modules return the values from the ROM. If the x coordinate is out of the range where the play map is located, they return zero.

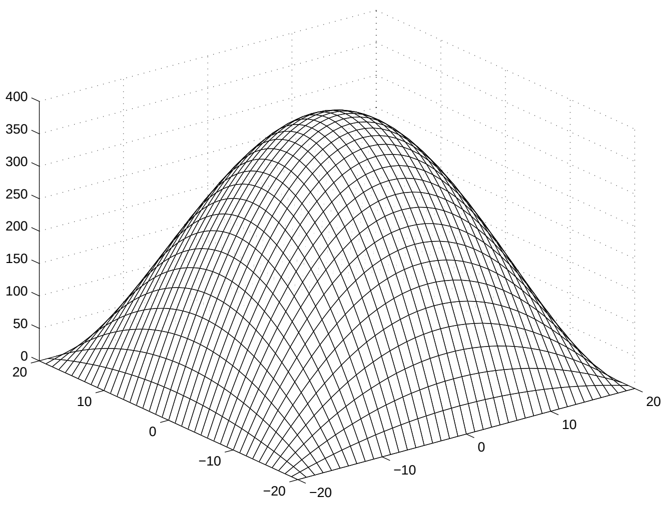

Figure 5: Background Parabola

3.4 Ball and Shadow Pixmaps

For the ball map, we stored a pixmap of the ball in the ROM. First, we offset the current y screen coordinate by the z coordinate of the ball. If the pixel was within the appropriate offset x, y screen coordinates, we return the data from the ROM. The actual data in the ROM has the left and right halves switched, so we can save an addition in hardware.

The shadow map (which is precalculated to look like a shadow of the ball, appropriately stretched for the race track) is stored in ROM identically to the ball ROM, but we omit the z coordinate logic.

3.5 Texture and Lighting

3.5.1 Background Generation

We wanted a background that consisted of concentric gradient circles moving in from the outside. However, due to size and speed limitations of the FPGA, a multiplier would be out of the question. We found we could do a reasonable approximation using a parabola through the color space, as shown in figure 5.

We can generate a parabola using just additions. In the X coordinate, for every pixel, we compute:

$$x_{n+1} = x_n + v_{x,n}$$

And

$$v_{x,n+1} = v_{x,n} - c_y$$

Where c_y is the curvature associated with the current y coordinate. We apply the same operation to make a parabola of curvature through the y coordinate:

$$c_{y=n+1} = c_{y=n} + v_{c_{y=n}}$$

And

$$v_{c_{y=n+1}} = v_{c_{y=n}} - 1$$

The initial vx is $c_y \cdot 320$, such that the velocity goes to zero towards the center of the screen. $320 = 256 + 64$, so this multiplication can be implemented as a single addition, with bit shifts.

We animate the texture by adding a the counter into the current color value.

This set of initial values generates squares that start towards the outside of the screen, and transform into circle towards the center. By starting with some initial curvature, and using the c_y parabola to set initial values, we could generate pure circles. However, we found we preferred the rectangle effect.

We implemented a test version of this as a C program on a computer, and used the test version to calculate appropriate minimum bit lengths.

As an alternative background, we selected the lower order bits of the parabola, which generates a fractal-like moire pattern. The user can change the background by switching modes.

3.5.2 Course Texture and Lighting

After experimenting with a number of textures, we found that textures with sharp lines were either very crude towards the front, or generated unpleasant moire patterns towards the back. As such, we used a texture consisting of gradient-colored squares. To generate the texture, we used the 9th through 15th bits of the translated x coordinate for red, the 9th through 15th bits of the translated y coordinate for green, and a constant for blue. To add lighting, we subtracted the inverse of the 9th through 3rd bits of the screen y coordinate. This made the race track darker towards the back. This corresponds to the situation of having the texture as an ambient color, with gray as the diffuse color. While this is not a very realistic scheme, due to the cost of multiplies, this seemed like a reasonable compromise.

3.5.3 Shadowing

Shadowing is accomplished by left shifting the bits of the course texture map over by one, if we are at both the level map and shadow map.

3.5.4 Ball Lighting

The ball texture is a gradient of blue, with intensity set by the y screen position. We could make a very realistic lighting scheme using the parabolic system used for background generation, with a circle of color centered near the top of the ball. This looked very realistic in computer simulations, and would have merely involved cut-and-pasting VHDL code. However, we did not have sufficient space on the FPGA to implement this.

The green bits are always zero. The red bits correspond to the current level of boost.

3.5.5 Transparency

Transparency is handled in the last stage of the pipeline. If there is a play map on the second level, and the ball is not in front of the play map, the bits of the current texture are right-shifted by one. The MSBs are set to $r = 0$, $g = 1$, $b = 1$. Otherwise, the color is fed through as-is.

3.6 Sync generation

The coordinate generator sweeps the x coordinate from 0 to 808, and the y coordinate from 0 to 524. The sync generator outputs a horizontal sync for $663 < x < 760$, and a vertical sync for $y > 491$. This is longer than specified by the VGA specification, as documented in the Altera UP1 manual. However, we found that if we followed the specification, some monitors failed to sync. If we increased the pulse width of the sync signal (but not the time off-screen), it worked on all monitors available for testing.

3.7 Stereoscopic Display

The game has a stereoscopic mode using LCD shutter glasses. On the half of the refreshes, the image is displayed to the left eye. On the other half, it is displayed to the right eye. LCD shutter glasses blank out the appropriate eye. Adding stereoscopic mode involved the following changes to the code:

- We added two outputs bits to the FPGA, corresponding to the lowest order bit of the game clock counter, and the inverse of that bit.
- If the lowest order bit of the clock counter is 0, and we are in stereoscopic mode, then the coordinate generation adds a constant offset to the translated coordinates, and another constant offset to the screen coordinates (corresponding to the appropriate offset at the location of the ball and shadow). We must also still keep the original screen coordinates for the sync generation.

We do not modify the shadow separation to be greater on the part of the shadow where it is closer to the user. This results in only a few pixels of error, and is not noticeable during game play unless one is looking for it.

3.8 Performance

We took a number of steps to optimize the pixel pipeline for performance. We have no multiplications, except by a constant factor, where the constant was chosen such that the multiplication could be expressed as a single addition, possibly with bit shifts. Most operations are tightly pipelined, and the architecture allows us to further add arbitrary additional pipelining, if need be, even if pipeline lengths do not match.

We are capable of running the pixel pipeline at the VGA dot clock of 25.175MHz. The next available clock speed grades available are 40MHz and 50MHz. We found that at 50MHz, the FPGA was incapable of performing a 32 bit addition in one clock cycle, and so it would be difficult to scale the architecture to this speed. There were significant glitches in rendering and game play at 40MHz, rendering the game unplayable. However, it appeared that it was probably possible to pipeline the code further to overcome these problems, although it would probably necessitate a larger FPGA. The game had no problems running at the VGA dot clock of 25.175MHz.

4 Sound

The audio portion of this project involved the generation, processing, and output of audio signals complimentary to the video game. The music was created in response to variations in the game such that the sound changed based on how the user was playing, through such parameters as the velocity and position of the ball.

The music consisted of three separate channels, each containing different components of the final audio signal, stored with 16 bits per sample and 44,100 samples per second. This allowed for the full range of audio frequencies with a high enough resolution to avoid the most of the unfortunate consequences of digitization.

The audio system received data from the video game system in order to control the mixing and filtering of the audio signals. Once the data had been received and latched out on the receiving kit, the appropriate control logic was used to decide how the scaling and mixing would occur.

On each output sample, 16 bits of audio data were passed out to a D/A converter, the output of which was connected to an audio amplifier and speakers.

4.1 Implementation Overview

The process described above was accomplished by having several pre-recorded audio signals stored in multiple ROMS, which were mixed according to multiplication factors generated by control logic within the system. The three channels consisted of the drums, melody, and harmony, and the varying of the scaling factors allowed for such occurrences as the drums fading in as the speed of the ball increased, and so forth.

The system was clocked at 10MHz, and the music sampled at 44.1KHz, yielding a rounded 227 clock cycles to generate each sample output. On each sample cycle, each of the ROMs had a turn driving the data bus to input the current audio sample to the system. The kits were synchronized so that as data was received from the video game kit, the parameters of the game were latched internally into the system when they became available. Control logic used the current parameters available to decide how the music should be mixed, and a filter applied to the coefficients was used to increase the precision of the coefficients and apply a low-pass filter to constrain the system from fading in and out too quickly.

4.2 System Evolution

The progress of this project progressed through several stages, as the feasibility of different implementations became realized. The first plan included lofty ideals of taking the DFT (Discrete Fourier Transform) of the weighted sum of the inputs in an attempt to filter based on the input ROMs. Unfortunately, as described below, size constraints of the 10K70 FPGA made this goal unattainable, and the limiting number of gates continued to pose a frustrating challenge in subsequent attempts at implementation.

4.2.1 FFT

A substantial portion of time was spent attempting to create and utilize an FFT (Fast Fourier Transform), first by using code supplied by `opencores.org`, and later by attempting an individual implementation. The code supplied by `opencores.org` was finally able to compile after the difficult battle of resolving the problem of the unsigned libraries used in the code that was not recognized by Altera. However, even after getting it to compile (or rather, after surpassing the 'compile netlist' section of the compilation process), the code was unable to fit in the 10K70 FPGA.

That being the case, the process was undertaken to create an FFT using a parallel implementation of a radix 4 DIF (Decimation in Frequency) FFT. Code was written to implement the butterflies and store the twiddle factors, and was getting to be quite sizable when further research was done on other implemented FFTs and the size required to implement them. After such investigation, it was discovered that at best, it looked possible to implement an 8-bit/sample 32 point FFT requiring approximately 16 clock cycles per point using 70,000 gates.

However, this was unacceptable for a number of reasons. To begin with, at sixteen clock cycles per point, there would be 512 clock cycles for the entire DFT which more than doubles the 227 clock cycles necessary for a 44.1kHz sampling rate, assuming a system clock of 10MHz. To exasperate matters, 32 points was an unacceptable precision considering the fact that we were dealing with audio signals which would be particularly sensitive to the levels of resolution. In addition, the 8 bits per sample would completely negate the use of 16-bits earlier in the system, which was desired for higher quality of sound. Lastly, even if it were decided to implement the FFT, there would be no room left on the FPGA for other portions of the project, thus further negating the purpose.

4.2.2 Subsequent Implementations

After it had been decided that a VHDL implementation of an FFT was not a feasible option, the next choice was an IIR filter with the coefficients of the filter changing based on the input parameters. The idea was to use a filter with 4 zeroes and 4 poles, with a system function of the form:

$$y[n] - \sum_{i=1}^{4} a_i \cdot y[n-i] = \sum_{i=0}^{4} b_i \cdot x[n-i]$$

This system was implemented, and compiled and simulated successfully, but gave warning errors regarding the size it was requiring in the FPGA.

After the first compilation, a trivial change was made regarding the width of an output signal, and after compilation, the system proceeded to stop simulating correctly. The file was changed back to its original state, and compiled again, at which point the simulation continued to give strange (but different) results. At this point, compilation was taking over half an hour, and upon suspicion that the simulation was changing without different files being compiled, several tests confirmed that even though the VHDL file stayed the same, after each compilation, the outputs produced were drastically different.

It was consequently decided that the filters would be implemented using a ROM of stored filter coefficients, and filtered according to an FIR system similar to that implemented in Lab 3. Once again, this tactic simulated encouraging results after the first simulation, and upon changing minor details and recompiling, the system failed to produce the meaningful outputs. The software continued to complain of space constraints, and the compiling time was almost 45 minutes and the simulation time around 15 minutes.

Upon the suggestion of experienced individuals, it was pondered that the VHDL code was taking up too much of the FPGA, and suggested that it would be advisable to scale back on the amount of space required for the system. That being said, it was decided to first implement the system without the middle filters, and add more of the filter elements piece by piece as the system continued to work.

This report contains the final implementation demonstrated at the time of the project checkoff, but also includes a description of other files which were necessary to cut out due to space and random FPGA flakiness constraints.

4.3 Storing Data in ROMs

The first step in implementing the system was to convert the `.wav` files created by fellow 6.111 student, Dan Roy, into meaningful data to store in the ROMs. This was done by reading in the `.wav` files into a vector by MATLAB, and writing a MATLAB file to take that input vector in decimal format from -1 to 1 and produce a data file containing 16 bits of signed magnitude numbers in hexadecimal format. This was done by multiplying the absolute value of each of the numbers by 2^{15}, and rounding. The signs were addressed by creating a vector of equal length containing the sign for each index (1 if the number was negative and 0 if the number was positive), and then multiplying that vector by 2^{15} and adding to the magnitude of the rounded number. This accounted for the sign bit, and since the magnitude of the number could not be beyond 2^{15}, there was no possible interference of the bits between the sign and the magnitude. Once computed, this number was then passed to MATLAB's `dec2hex` function, which converted the positive decimal integer number to a 16-bit hexadecimal number. The resulting vector was output into a comma separated value file, and a `perl` script was written to take such a file and produce a meaningful `.dat` file. Since the ROMs used were only 8 bits in width and our data was 16 bits wide, it was necessary to split the audio data into the higher and lower order bits, and program each ROM with the appropriate bits of the sound file. Thus, the `perl` script took as an input file the CSV file created by MATLAB, and output two `.dat` files suitable for programming a ROM.

4.4 Current Implementation of Audio Processing System

The implemented system worked as follows: on every input sample, the audio data was read in from the ROMs and scaled by the scaling factor determined by the control system, based on input parameters from the video game. These scaling factor coefficients were themselves low-pass filtered to control the fade-in and fade-out times to ensure that the system was less susceptible to sudden changes in the game. In other words, even though the parameters in the game were likely to change suddenly, since it was established that the aesthetic value of

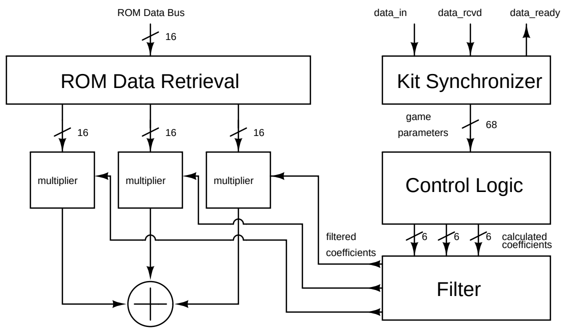

Figure 6: Overall Block Diagram of Audio System

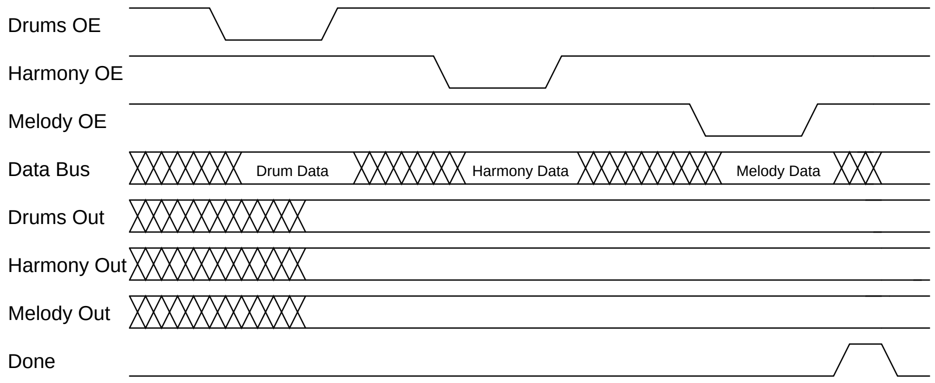

Figure 7: Timing Diagram for ROM Inputs

the sound deteriorated when allowed to change suddenly, it was deemed necessary to low-pass filter the coefficients to put a limit on how quickly those coefficients could increase or decrease. Once scaled, the audio inputs were summed, and then output to the DAC in the appropriate format.

4.4.1 ROM Inputs

Since each of the 16-bit ROM inputs were unable to have their own data bus, it was necessary to keep track of which one was driving the bus at any particular time to avoid bus contention. Thus, on each sample cycle, each of the ROMs cycled through their turns driving the bus, and was subsequently latched to the appropriate output. A VHDL file, `romsync.vhd` was created to control the ROM accesses, taking the 16 bits from the ROM data bus and producing 3 valid sets of each of the data, and outputting a signal to indicate completion at the appropriate time. A timing diagram is shown in figure 7.

4.4.2 Sample Rate Generation

A function samplerate.vhd was used to easily control the sample rate based on the clock input. Since the signals were to be played at 44.1kHz, it was necessary to count the clock input 227 times (assuming at 10MHz clock) to output samples at a rate of 44.1kHz. A sample signal was output as high once every 227 clock cycles.

4.4.3 Multiplication

The multiplication was done via a shift and add approach in signed magnitude format, shifting the the input by the appropriate number of bits as deemed by the second input, `AND`ing it when with the corresponding bit, and then summing the results. The most significant bit, which indicated the sign, was determined by `XOR`ing the highest order bits of each of the input.

4.4.4 Control Unit

The control unit generated output multiplication scaling factors based on the parameters in the game. Since it was established that the y velocity (velocity going into the monitor screen) would be the most appropriate indicator of the overall speed of the game, that input parameter was used to make most of the decisions regarding scaling factors. The coefficients were stored as 6 bit positive numbers, although it was quickly found that a number higher than 16 resulted in aesthetically displeasing music. Consequently, the control unit was engineered to produce scaling factors for each of the audio signals less than or equal to 16.

Since it was desired that there be music at all times, the melody multiplication factor was kept constant, allowing the other two channels to fade in and out as the game progressed.

The multiplication factor for the drums was chosen to be the absolute value of the y velocity divided by 16 (bit-shifted by 4). Thus, the relative magnitude of the drums' presence was proportional to the speed at which the user was playing.

The multiplication of the harmony was dependent on the the z velocity (vertical velocity), and was once again proportional to the speed.

4.4.5 Coefficient Filtering

As mentioned previously, the multiplication coefficients for each of the ROM inputs was created by the control unit. However, this allowed the coefficients to change much too drastically than was audibly pleasant. Consequently, a filter was created that low-pass filtered the coefficients to prevent them from changing too quickly.

However, since the sampling rate was so high, it was necessary to effectively decrease the sampling rate with regards to how quickly the filter coefficients could change so as to allow for a lower level of precision on the coefficients of the filter. Since a first order single-pole IIR filter of the form $y[n] - a \cdot y[n-1] = b \cdot x[n]$ was used to filter the coefficients with b being very close to 0 and a being very close to 1, the higher the sampling rate, the more bits required to implement the fractional component of a. Thus, in the coefficient filtering process, a counter was used to keep track of the number of samples which had elapsed to ensure that the coefficients could only change every 600 samples, or approximately 74 times per second. Consequently, a fraction of $\frac{31}{32}$ was sufficient for a and $\frac{1}{32}$ sufficient for b.

The fractions were implemented by extending the bits of the input coefficients by four, and first multiplying the previous output by 31, adding it to the current input (bit-extended), and then shifting the bits to effectively divide by 32. The output was stored in 15 bits, but

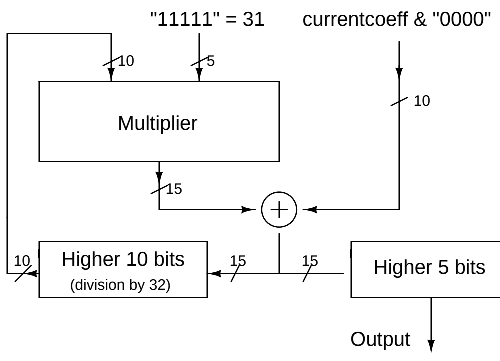

Figure 8: Block Diagram for Coefficient Filtering

only the upper five output as the subsequent coefficient. Refer to figure 8 for the block diagram, and figure 9 for a plot of the transfer function.

4.5 Testing and Debugging

As mentioned previously in this report, the testing and debugging of this system led to the drastic simplification of logic and implementation to allow for ease of fit into the FPGA and to submit to the FPGA's seemingly random whims. The process of testing and debugging, and the overall evolution of the system have been thoroughly documented in previous portions of this report.

5 Game Play

The game play logic is both complicated and uninteresting. On each refresh, it increments x by the x velocity vx. It increments y by vy. If the user is holding an arrow, it increments or decrements vx or vy. If the user is on a platform and hits jump, it sets vz to some positive value from the higher-order bits of the *boost* variable. If the user is above a platform, and would be below it if the z coordinate was incremented by vz, it sets the z coordinate to that of the platform. Otherwise, it increments z by vz, and decrements vz. If the user falls below 0, it resets x, y, z, vx, vy and vz to 0.

If the user holds down the boost button, and does not hit the jump button, the *boost* variable increases, until it reaches some maximum value. When the user releases the boost button or hits the jump button, it falls back to a minimum.

In addition, one of the remaining buttons can be used to switch modes (four possible modes, with one bit selecting background, and the other, stereoscopic mode). The other one sets the game into paused/screenshot mode.

The joystick buttons are not synchronized. They are read once every 30th of a second, for a time of at most $\frac{1}{25.175 \cdot 10^6 Hz} \approx 0.04 \mu S$. The odds of reading while the joystick is changing are less than 1 in 800,000 per key press. This is not a dominant failure mode for the system.

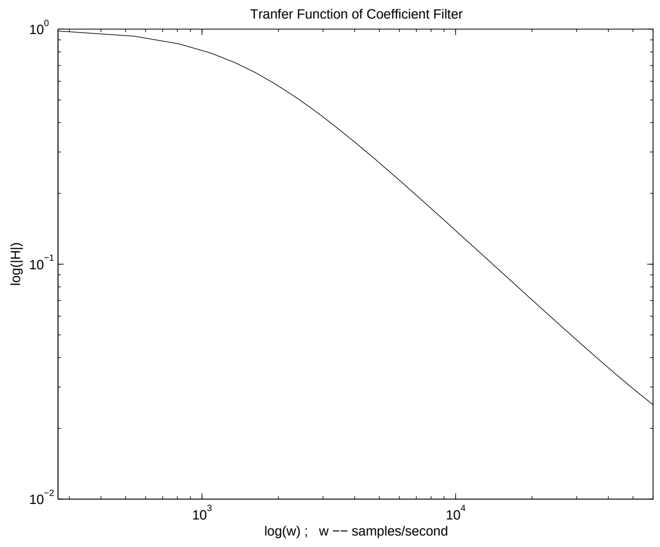

Figure 9: Transfer Function for Low-Pass Filter

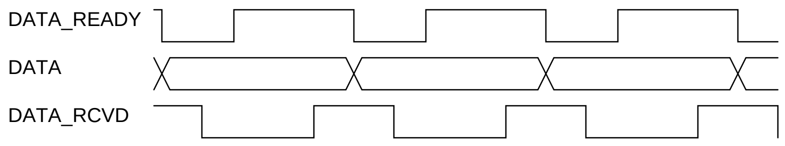

Figure 10: Timing of Interkit Communications.

6 Communications

Communications is implemented through a one bit parallel port interface. The timing is shown in figure 10.

The transmitting kit puts data on the data pin, and signals `data_ready`. When the receiving kit receives the data, it signals `data_rcvd`. At that point, the transmitting kit lowers data ready, and puts new data on the bus. The receiving kit acknowledges by lowering `data_rcvd`. At that point, the transmitting kit raises `data_ready`, and the cycle repeats.

6.1 Transmitter

On the transmitting side, this is implemented with FSM shown in figure 11, with states:

State	`Data_ready`
`idle`	0
`data_rcvd`	0
`next_data`	0
`data_ready`	1

And the data output bit changing on `next_data`. The system enters the idle state once it

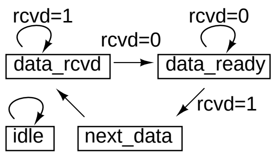

Figure 11: Transmitter FSM.

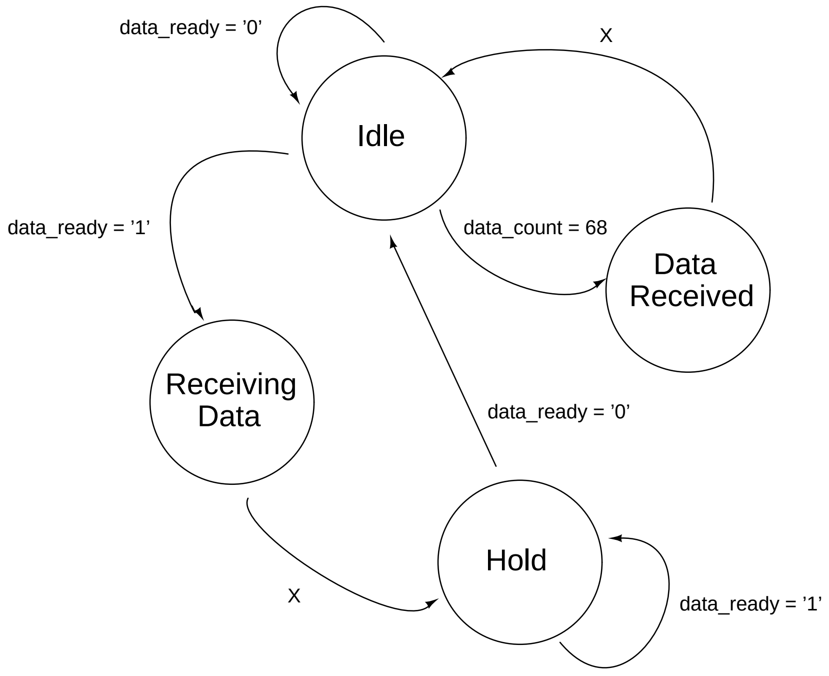

Figure 12: Kit Synchronization FSM

has transmitted all the data, and enters the `data_received` state when new data is available (at the time of the screen refresh).

The `data_rcvd` pin is synchronized on the rising edge of the clock. We protect the FPGAs from programming errors by adding 510Ω resistors in series between all pins connecting the kits.

6.2 Receiver

The FSM for the receiving kit is shown, and as can be seen, the system began receiving data when the `data_ready` signal went high, and after 68 bits of valid data, latched the data out in a parallel fashion. If the receiving system remained idle for longer than 6 clock cycles, the system automatically reset itself, assuming that the full packet of 68 data bits were not received properly.

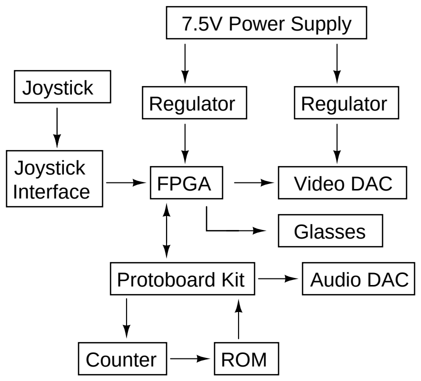

Figure 13: Wiring Block Diagram

7 Wiring

Due to the 25.175MHz VGA dot clock, the high-speed DAC, the power converter and the general unreliability of protoboards, the 3D engine and game play portion of the project was constructed through soldering and wire-wrapping. The main digital board was wire wrapped (with the exception of the input voltage regulators), while the joystick interface, the video DAC board and the power converter were soldered. The sound system was constructed on a standard protoboard kit. The overall wiring is shown in figure 13.

The regulators are standard LM7905 chips. All chips have 0.1uF bypass capacitors (not shown in diagrams). All rails have large electrolytic bypass capacitors (also not shown).

7.1 Joystick Driver

A PC joystick behaves as a potentiometer to V_{DD}. The ideal way to drive this is with a current source, since we get a voltage linearly proportional to the joystick offset. To save on component count, we instead used a pull-down resistor, and compared the output voltage to a voltage reference.

It is trivial to show that the maximum split between the 50kΩ and 100kΩ happens when the pull down resistor is $100/\sqrt{2}$k$\Omega \approx 70$kΩ. Since we found some variety in how joysticks are calibrated differently, we left the comparison voltage as a trimmer pot.

The buttons are pull-downs to ground. We put in a pull-up resistor to set the output to default high.

To protect the FPGA from damage due to bad programming, we put in a protection resistor in series with the outputs.

This gave eight outputs (left, right, up, down and four buttons), with the circuit as shown in figure 14.

7.2 Stereoscopic glasses

The stereoscopic glasses did not start blanking until a little above 10V, and did not fully blank until reaching a little above 12V. Our board only had a 5V power supply. As such, we

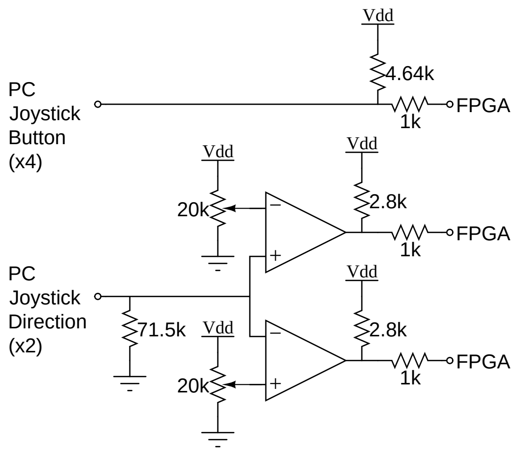

Figure 14: Joystick Interface Circuit

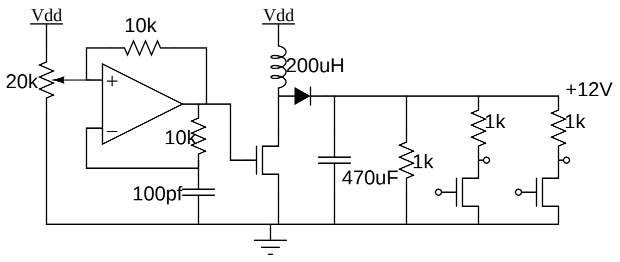

Figure 15: Power Converter and Shutter Glass Interface

needed an up-converter to drive the glasses. We decided on a standard continuous conduction boost converter topology, since its load independence, combined with the lack of need for accuracy in the boost converter, allowed us to step up the voltage without need for a control network, reducing component count. The circuit is shown in figure 15.

For the video DAC, we used the Intersil CA3338 chip. This chip is capable of directly driving a video output without need for further amplification. We calculated the appropriate network to have a VGA-standard 0.7V output voltage, and 75Ω output impedance. The circuit is shown in figure 16.

The circuit was constructed on a breadboard with solder pads on one side, and a ground plane on the other.

8 Conclusion

We built and demonstrated a simple 3D racing game entirely in hardware. Overall, we felt the project was a qualified success. In retrospect, we might have made a few minor modifications:

- We would rely on bit shifts less. We used them extensively throughout the architecture under the (faulty) assumption that they were free in hardware. However, due to the

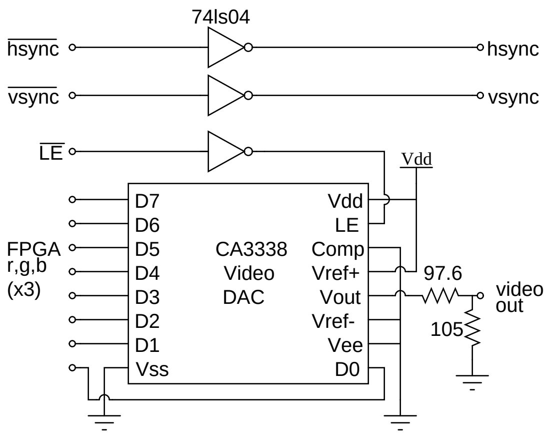

Figure 16: Video DAC and Sync Driver

nature of the Altera FastTrack interconnect architecture, they were in reality fairly expensive. We ran out of interconnect well before we saturated number of gates.

- We would begin implementing with a faster clock speed, for a higher refresh rate. When we began, we only had a 25.175MHz clock available. It was not feasible to retrofit even a 40MHz clock without substantial architectural changes.

- Antialiasing would require a multiplier. With a slightly larger FPGA, we could fit a pipelined combinational multiplier. We would have to store the previous pixel value for horizontal pipelining. For vertical, we would have to store an entire row of values. The appropriate values to multiply the colors by would be the lower order bits of the translated coordinate.

References

[1] Altera. *University Program Design Laboratory Package User Guide* 1997.

[2] Oppenheim, Alan V., Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing* 1999: Prenctice Hall.

[3] Babel. Website: http://www.ufoot.org/babal/ Accessed May 2003.

[4] SkyRoads. Website: http://www.blueroom.ee/history/skyroads Accessed May 2003.

[5] Chandrakasan, Anantha, Rex Min, Manish Bhardwaj, Seong-Hwan Cho, and Alice Wang *Power Aware Wireless Microsensor Systems* 2002: Keynote Paper, ESSCIRC, Florence, Italy.

[6] Troxel, Donald E. *Serial Interfaces for Minicomputers* 1975: IEEE Transactions on Computers 24(10): 1027-1028.

[7] Landolt, O. , A. Mitros and C. Koch *Visual Sensor with Resolution Enhancement by Mechanical Vibrations* 2001: Proceedings 2001 Conference on Advanced Research in VL.SI

[8] Mitros, J. C. et al *High-Voltage Drain Extended MOS Transistors for 0.18μ Logic CMOS Process* 2001: IEEE Transactions on Electron Devices Volume: 48, Issue: 8.

9 Appendix A - Color Plates