Rave in a Box

Sammy Cherna, Josh Gruenstein & Matt Reeve

6.111 Project Report, Fall 2018

Abstract

This report describes the conception, design, and implementation of *Rave in a Box*, an FPGA-based audio-responsive laser projection system. The box takes in live music and performs Fourier-based signal processing methods to identify peaks of structural novelty. It then uses a laser and set of galvanometers to project animated vector graphics onto a nearby surface in time with transitions in the music, such as key changes, transitions from verse to chorus, and introduction of new instrumentals.

Contents

1 Overview

FPGAs are uniquely powerful tools for live signal processing and real-time control. Their tight timing capacity and reconfigurability allow these categories of computation and IO to occur with a far lower power budget than microcontrollers or other embedded systems.

For our 6.111 final project, we sought to capitalize on these two unique capabilities by having an FPGA generate a live laser light show in response to a musical soundtrack. At the highest level, our design can be summarized via the following block diagram:

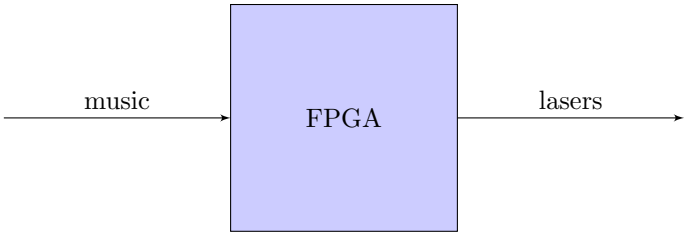


Figure 1: A macro view of the Rave in a Box.

Our box takes in audio via the Nexys4 DDR's ADC, performs signal processing and graphics generation, and outputs analog galvonometer control signals and delayed audio through a set of DACs. The galvanometers have mirrors glued onto their axles which reflect a laser beam according to their angles. We then exploit persistence of vision to project shapes by cycling through a path at high speeds.

The work necessary to achieve this can be broken down across signal processing, graphics generation, and IO, in that order of complexity. We will discuss the algorithms we utilized and developed for each of these domains, and specific details as to their implementation in hardware for the FPGA.

We began this project with only an abstract conception of our goals, and only through iterative research and development arrived at a final working device. We hope to communicate that process of discovery propelled by a love of music and bright shiny lights in this report.

2 Project Logistics

We divided up the responsibility for the project as follows: Sammy Cherna was responsible for the signal processing subsystem. Josh Gruenstein was responsible for the graphics subsystem. Matt Reeve was responsible for all of the hardware and the hardware control subsystem. Despite these delineations, all

three of us collaborated heavily on all parts. The three of us live together, so collaborating together was natural.

Our original goals for the project were as follows. For our minimum project goals, we wanted to compute the spectrogram of incoming audio, and based on a feature of that spectrogram, display a certain static image (a frame) with the laser galvonometers (controlled via DAC over SPI). Each frame would consist of instructions representing line segments to interpolate between. For our standard project goals, we wanted to compute the spectrogram and chromagram of incoming audio, and based on a feature of that chromagram (for example, the most prominent pitch class), display a certain animation of images (a scene composed of frames) with the laser galvanometers. We had originally had many stretch goals, which can roughly be broken down as follows: more advanced graphics generation (such as interpolating along Bezier curves instead of lines), more advanced signal processing (using Hanning window for better FFT, further processing on the chromagram to get more meaningful graphics selection), implementing tempo analysis and incorporating tempo into graphics, and more advanced hardware (potentially using 3 different colored lasers and combining them with optics).

While we did not have time to tackle tempo analysis or multiple colored lasers, we accomplished all of our other stretch goals, aside from all of our standard project goals, including one stretch goal that we did not even imagine: song segmentation / structure analysis. We realized that merely selecting graphics based on the most prominent pitch class in the chromagram would only yield pleasing results on very simple synthetic examples, and would not work well on actual songs. Consequently, we decided to tackle the huge endeavor of song segmentation by creating our own real-time adaptation of a song segmentation algorithm, the first of its kind.

Aside from the song segmentation, we implemented our stretch goals of using a Hanning window for better FFT results, interpolating along Bezier curves for advanced graphics (as well as implementing a system to convert raster images to vector graphics for the laser galvos), and creating a sturdy yet stylish custom box for our system.

3 Signal Processing

Music Information Retrieval, or MIR, is the exciting interdisciplinary study of extracting useful features from musical data. Part of our motivation to pursue a project in MIR stems from one of our members (Sammy Cherna) taking 21M.387, Fundamentals of Music Processing. Many of the processes used in our box were adapted from material taught in that class.

In constructing an MIR system for our box, a major challenge we encountered was adapting algorithms traditionally implemented in software and run

on stored audio samples to a live system implemented in hardware. Thus, in this section we will discuss our signal processing in three steps: the fundamental algorithms we used, the modifications and compromises we made to fit our application, and our module-level implementation in hardware.

3.1 Algorithmic Approach

In the beginning, there were raw audio signals.

Our box takes incoming music as analog signals delivered via an aux cable, then transcribed into digital data and delivered to the FPGA via the Nexys4 ADC. However, there is limited advanced processing that can be done on raw audio data alone. Instead, nearly all approaches first run audio through a *Fast Fourier Transform* to produce a *spectrogram*.

This requirement stems from the fact that the fundamental building block of audio signals are waves of different frequencies and amplitudes. In order to determine useful information about a given signal, we must determine its composition across different frequencies. This representation is called a spectrogram, which can be computed by running an FFT across a sequence of audio signals.

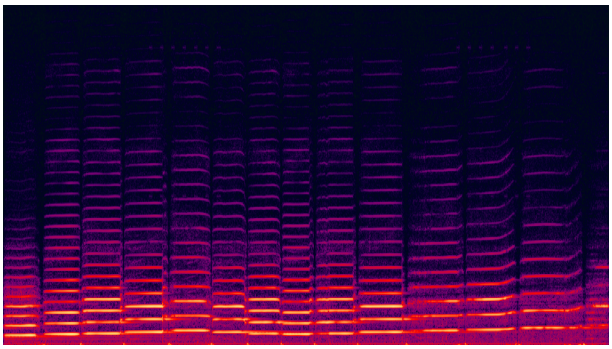


Figure 2: Spectogram from Wikipedia of a violin recording. On the horizontal axis is time, and on the vertical axis is increasing frequencies. Additional bands of intensity are from harmonics.

The spectrogram tells us how much fundamental waves of different frequencies contribute to the signal at a given time. For example, if you were to play a middle C on a piano and generate a spectrogram from the recorded audio, you would expect to see high intensity at ∼262hz, in addition to slightly lower intensity at overtones 524hz and 768hz, overtones and harmonics one and two octaves away from middle C. Thus the spectrogram is far more informative than raw audio, which is difficult to interpret without additional processing.

In addition to the spectrogram, we can also compute the *chromagram* of

audio by summing all of the harmonic frequencies of each of the 12 Western pitch classes: C, C♯, D, D♯, E, F, F♯, G, G♯, A, A♯, and B. This allows us more easily to determine the note composition of audio (the piano example from above would clearly be far easier to classify).

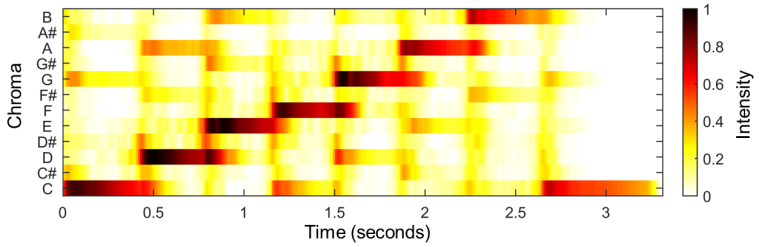


Figure 3: Chromagram from Wikipedia of a C major scale played on a piano.

Our original plan was to continuously compute the chromagram on incoming audio, and use the highest intensity pitch class to select a graphic to project. However, experimentation is software demonstrated that this approach on most music would not yield scene transition timing that would make sense to a listener. Chromagrams of popular music are often very noisy and fast-changing due to the presence of many instruments and tracks.

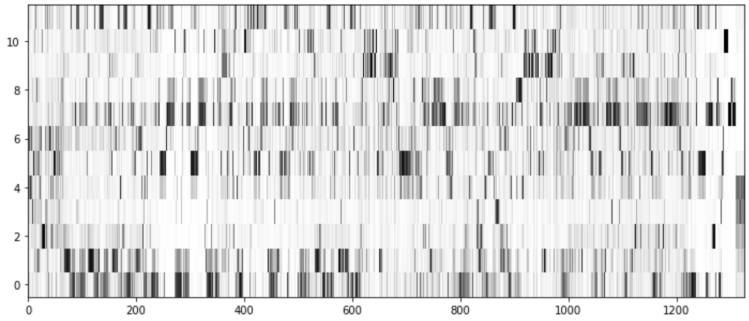


Figure 4: Chromagram of "The Bends" by Radiohead computed by our team.

While an approach like this would certainly be of sufficient technical complexity, and work well for some recordings, research told us that raves rarely play pure sin tones or classical piano pieces. Thus, this methodology would probably be insufficient for our goal of creating a **true** Rave in a Box.

What we needed was an algorithm for *song segmentation*. Unfortunately, song segmentation is still an open research problem, with a diverse array of proposed solutions but no definitive methodology. Additionally, nearly all methods operate on the entire song at once, and are extremely computationally intensive.

One common theme in recent song segmentation papers is the computation of a two-dimensional self-similarity matrix, first utilized in [1]. In these approaches, a song chromagram matrix is multiplied by its transpose to produce an $t \times t$ matrix, where t is the number of columns in the chromagram. As the dot product between two vectors is analogous to their covariance, each cell in the self-similarity matrix is proportional to the correlation between samples at times corresponding to the row and column that cell is in.

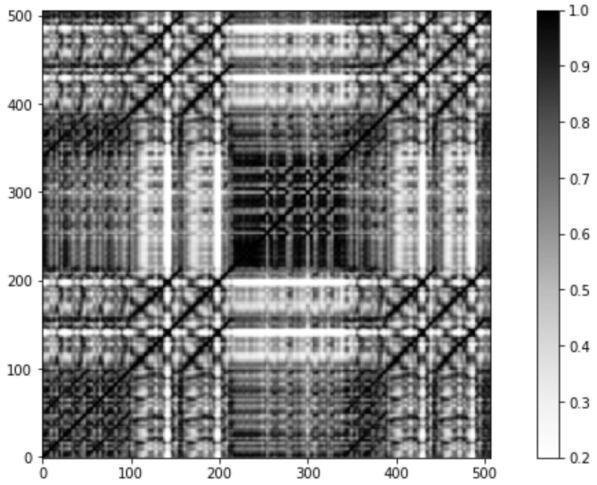


Figure 5: Self-similarity matrix of Brahms' Hungarian Dance computed by our team. We can see that square-shaped regions are regions of high homogeneity, while corners between them represent strong structural changes in the song.

From the self-similarity matrix we can compute some score of *structural novelty* by applying a checkerboard kernel to its diagonal. If the kernel is centered at sample t, it adds correlation values on the same side of t, and subtracts those on different sides. Thus, the value is always greatest if the two halves of the music sampled by the kernel are similar to themselves but different from each-other, making it a successful measure of novelty.

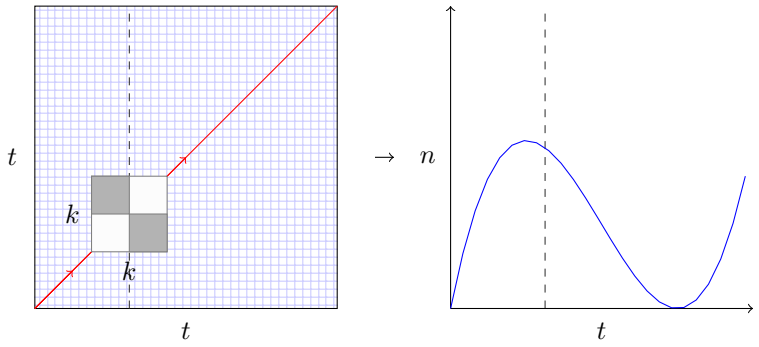


Figure 6: Application of a $k \times k$ checkerboard kernel to a $t \times t$ self-similarity matrix to compute a structural novelty curve $n(t)$.

This continuous curve of song novelty over time allows us to identify major structural changes (and thus ideal times to change graphics for our box) by identifying peaks in the novelty curve. We implemented this system in Python, and found that it was succesful in identifying good scene transitions for a wide variety of music.

3.2 Modified Algorithm for Hardware

The method of computing structural novelty based on a self-similarity matrix presents numerous challenges for implementation on an FPGA. At first glance the algorithm requires precomputing scene transitions on entire songs, which would likely exceed the Nexys4's memory capacity. Furthermore, while trivial in software, implementing large matrix multiplication pipelines and kernel applications in Verilog can be difficult.

Through a change in algorithmic perspective, we were able to implement a fully functionally equivalent system with far less computational and memory cost. This stemmed from the recognition that we did not need to compute the entire self-similarity matrix, but instead directly calculate the kernel value at any point in time. We achieved this by storing k (the kernel size) chromagram samples in a FIFO, and at each new chromagram updating the novelty score to reflect the updated FIFO.

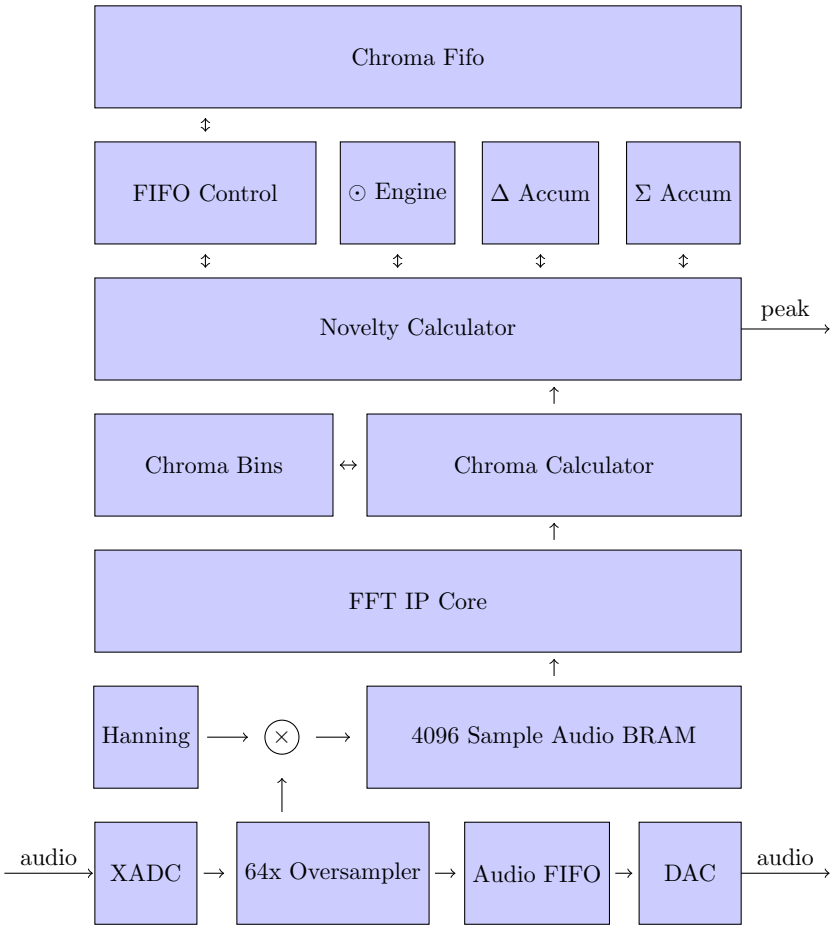
To accurately reflect the value of the kernel in the original algorithm, at any point the novelty must equal the dot product of each pair of chromagrams in the FIFO, where pairs are added if they are from the same half of the FIFO and otherwise subtracted. However, to recompute this value from scratch would be unnecessary computation, as at any time-step only three sets of dot products change: those involving the new chromagram, those involving the last, and those involving a chromagram in the center of the FIFO which previously was

on one half and just moved to another. By only computing these three sets of dot products and applying them as deltas to the previous novelty score, we can do $O(k)$ rather than $O(k^2)$ computation at every new chromagram.

While this method does entail storing k chromagrams and $k/2$ chromagrams worth of audio (to replay in synchrony with the computed novelty curve), this is far less memory intensive than storing an entire song's worth of both audio and chromagrams.

Interestingly, from a literature review we believe we are the first to create any sort of live song-segmentation algorithm, let alone implement one in hardware.

3.3 Module-level Implementation



Our Verilog for signal processing began with the sample FFT code provided for the class by Mitchell Gu. This included creating a 104MHz clock, sampling the onboard ADC at 1MSPS, oversampling by 16x to get 14-bit samples at 62.5KHz, storing 4096 of them in a circular BRAM, feeding them to the FFT Mag IP Core to get the magnitude (square root of real part squared plus imaginary part squared) of the FFT, storing the results in a BRAM, and finally displaying a histogram of this data (the spectrogram) on the VGA output. In order to get this code to work, we had to splice open an AUX cable and apply a DC bias to the signal (configurable via potentiometer) for it to be read properly by the ADC. One issue we encountered was clipping from the ADC, despite the signal staying in the acceptable 0-1V range, so we lowered the volume of the input music until we did not have clipping. We decided to modify this code by oversampling by 64x instead of 16x, so that we could get 15-bit samples at a rate of 15.625KHz. There were three reasons for this change: keeping a lower sampling rate would give us better frequency resolution (about 4Hz per bin in the FFT), it would let us look at a larger window in time for each FFT (about 0.26 seconds) which would help the novelty calculation be more robust, and the extra bit of precision could help for further calculations.

We also decided to pursue one of our stretch goals by incorporating a Hanning window into the signal in order to get better FFT results. The standard process of taking 4096 samples at a time from a signal consists of multiplying the theoretically infinite signal by a rectangular window in time. This results in spectral leakage in the FFT output, which can make computation like note detection less accurate, especially when accumulating many frequency bins into chromagram bins. In order to reduce spectral leakage, we can multiply the 4096 samples by a different window shape, such as the Hanning window, derived from a sin wave. We computed 4096 16-bit samples of a single Hanning window, and stored them in a ROM. When a new ADC sample would be ready to store in the circular BRAM, we would use its index out of 4096 to fetch the correct Hanning value from the ROM, multiply it by the Hanning value, and right-shift it by 16 before storing the 16-bit result in the BRAM. The ROM had a latency of 2 clock cycles, so we had to pipeline accordingly.

At this point we had working spectrogram computation. Our next step was to turn this spectrogram into a chromagram. In order to create a chromagram, we needed to sum up many different frequency bins for each pitch class bin. Since every frequency bin contributes to at most 1 chromagram bin, we decided to create a ROM which would take in a bin index and give us an integer 0-11 representing the chromagram bin that it corresponds to, or 12 indicating that it corresponds to no chromagram bin. This ROM was 4 bits wide (to represent the integers 0-12), and contained 1024 addresses, since we only cared about frequencies in the first 1024 spectrogram bins. As new outputs came from the FFT module, we would check its index using the tuser output, look up the chroma bin, and add it to the correct chroma bin. Only after receiving all 4096 outputs would we scale down each chroma bin and update the output from the chroma module.

Since our frequency resolution was only 4Hz, we did not want to consider octaves with notes less than 4Hz apart. Additionally, while almost all software MIR tasks include normalization of each chroma vector for increased robustness, we could not perform meaningful normalization as division is very difficult in hardware. Accordingly, we wanted roughly the same number of spectrogram bins to contribute to each chromagram bin, so that each is roughly on the same scale. We decided to only consider notes C3 (130 Hz) through E7 (2637 Hz) for chromagram contribution. While we could not perform normalization, we found that this chromagram computation was sufficient for our needs. We modified the spectrogram VGA module to display a histogram of chromagram intensities as well, so that we could observe our chromagram calculation in action.

Following the chromagram calculation for an entire window of samples, we pushed this new chromagram onto our 32-chromagram FIFO and computed the new novelty score for this point in time. As described above, instead of computing every possible dot product between pairs of the 32 chromagrams in the FIFO, we would keep a running accumulator of the novelty score, and only compute the dot products necessary to calculate the delta from the previous novelty score. This would require 3 sets of dot products, each one requiring a full pass through the FIFO. To simplify the process, we wrote a FIFO Controller module to interface with the FIFO, and a Dot Engine module to compute dot products. Unfortunately, both of these modules required significant debugging in order to get correct operation. For the FIFO Controller module, we had to ensure correct timing of the read and write lines in order to cycle properly through the FIFO. For the Dot Engine module, we had to properly pipeline in order to leave time for the 12 16-bit multiplications and subsequent 32-bit additions.

Once we got the FIFO Controller and Dot Engine working, we created a Delta Accumulator to properly accumulate all of the dot products without overflow or underflow. Each dot product result would either be added or subtracted from the accumulator, depending on the indices of the two chroma in the FIFO. The Novelty Calculator module then operated with the following state machine: push new chroma onto FIFO while storing the old chroma leaving FIFO, compute the dot product between the new chroma and all other chroma in the FIFO (by cycling through the FIFO), adding/subtracting from the Delta Accumulator accordingly, compute the dot product between the old chroma and all other chroma in the FIFO, cycle through the FIFO to retrieve the middle chroma (in index 16), compute the dot product between this middle chroma and all other chroma in the FIFO, and finally take the accumulated delta and add it to our running novelty score accumulator.

Amazingly, after an eternity of debugging, we were able to analyze the results of this novelty computation with Vivado's Integrated Logic Analyzer and see clear peaks in the novelty score at key transitions in song structure. In order to detect these peaks properly, we first implemented a simple low-pass filter by computing an exponential moving average of the novelty score. The current

filtered novelty would be 0.5 times the new computed novelty plus 0.5 times the previous filtered novelty. This helped smooth out some of the bumps and extraneous peaks in the novelty curve. Then we store three novelty values: the new one, the one from one timestep ago, and the one from two timesteps ago. We can then declare the one from one timestep ago a peak if it is greater than both the new one and the one from two timesteps ago. We also check that it is greater than a certain threshold before declaring it a peak. This resulted in quite accurate peak detection for certain songs, giving us pulses exactly at transitions from verse to chorus and vice versa. Unfortunately, due to our lack of normalization at various points in our computation, a good peak threshold for one song might not be good for a different song, and so our approach's robustness was limited.

The output from the Novelty Calculator Module, indicating if there is a peak in novelty at the current timestep or not, is fed to the Graphics module, which would trigger a scene change on a peak. However, since we expect a peak in novelty when the chroma belonging to a structural transition in the song is halfway through the 32-chroma FIFO, there is about a 4 second delay between inputting audio and outputting a peak. In order to remedy this, we also store a FIFO of audio samples corresponding to 4 seconds of audio (65536 15.625Khz samples), before outputting the buffered audio to a DAC over SPI.

4 Graphics

The Rave in a Box generates graphics by following a path and turning a laser on and off along it. Thus, for the box to be able to project graphics and maintain persistence of vision without obscene memory usage, it must be able to interpolate along some compact representation of vector graphics that travel the shortest possible path to across those graphics.

This challenge can be broken down across two domains: the generation in software of a shortest path of Bézier curves, and the interpolation in hardware across these curves.

4.1 Path Generation

Our team originally planned to generate line-based drawings by hand, which would likely have near-optimal path length due to human intuition. However, we quickly realized that even for 4 scenes, each with 16 frames, this would take an inordinate amount of time. Thus, we sought to find a method of automatically generating scene paths from graphics found online, most of which being in raster form.

We built the following software pipeline to intake animated GIFs and output .coe files containing Bézier curves and laser on/off instructions:

1. Split animated GIF into individual frames, and mask to black and white.
2. Trace a set of Bézier curves around each item in each frame.
3. Run a nearest-fragment greedy algorithm to find the ideal set of paths to connect objects in frames, and refine with simulated annealing two-opt.
4. Split the largest Bézier curves in half until there is a power of 2 number of curves in each frame.
5. Pack frames into a .coe file and output correct Verilog parameters.

Step 3 of this process is NP-Complete, as it is a close relative of the Traveling Salesman Problem which is also NP-Complete. Thus the generation of scene .coe files can be somewhat time consuming. However, we found this was necessary to create short enough paths to allow persistence of vision with complex graphics. The details of this process can be found in the Python code in Appendix B.

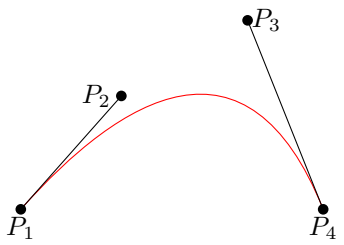


Figure 7: Example cubic Bézier curve with four control points.

Practically, the ROM described by the COE file is addressed by $\lceil \log_2(|S|) \rceil + \lceil \log_2(max(|F|)) \rceil + \lceil \log_2(max(|I|)) \rceil$ bits, where $|S|$ is the number of scenes, $max(|F|)$ is the maximum number of frames per instruction, and $max(|I|)$ is the maximum number of instructions per frame. Each line of the ROM corresponds to a cubic Bézier curve with four control points (and thus eight 12 bit numbers) and an additional bit to indicate whether the laser should be on or off.

4.2 Hardware Implementation

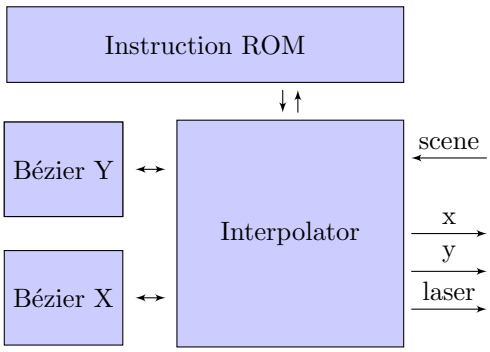


Figure 8: Block diagram of graphics generation subsystem.

The Interpolator module cycles through instructions in the provided scene address, and outputs instruction coordinates to two combinational modules that interpolate along the Bézier curve. It then forwards those outputs out to the SPI module, which in turn exports them to the DAC. This design requires no handshaking or clock-sharing with other modules.

5 Hardware

Our physical set-up consisted of a laser galvanometer set which included two galvanometers with mirrors on an aluminum mount, motor driver boards, a power supply, and a 5mW red laser. The motor driver boards each took in analog voltage inputs to control the galvanometers. Due to the fact that the Nexys can only output digital signals, MCP4822 two channel digital-to-analog converters were used over SPI to communicate with the motor driver boards. We also used a MCP4822 to output buffered audio, as PWMing audio out at our relatively low 15.625hz sampling rate yielded noticeably poor audio quality. A bipolar amplifier circuit was built and used to increase the overall laser projection angle (and thus picture size) and allow configurable offsets and gains for the x and y axes.

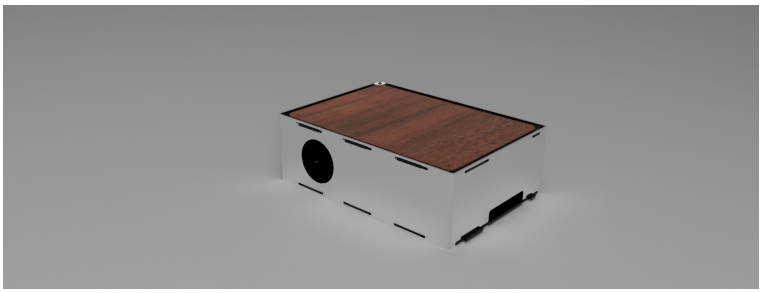


Figure 9: Rave in a Box 3D render from Autodesk Fusion 360.

Manufacturing of the project enclosure was an extensive process. First, the product was designed in computer aided design software to ensure fitment and appearance. Thereafter DXF files were able to be made in order to waterjet and laser cut parts. Laser cut eighth inch acrylic sheets, separated by aluminum standoffs were used as the frame of the box. A sixteenth inch Aluminum sheet was waterjet and formed around the outside of the frame before finally being brushed with steel wool for a textured appearance. A plywood sheet was laser cut and acrylic letters were laid in the sheet for the top cover of the box. It was then stained for a darker aesthetic.

6 Lessons Learned

After many many hours of grueling debugging and toiling over small mistakes, we have learned many lessons. We believe that the most significant lesson that we can share is to never assume that a module is working, despite how simple it may seem. Instead, it is crucial to validate the correct operation of every small module before moving on to other modules that use them. We used Vivado's Integrated Logic Analyzer to help us validate and debug modules, particularly ones sensitive to timing issues, and we highly recommend future 6.111 students do the same. We also learned how useful it is to utilize the module abstraction and create small sub-modules for individual repeated tasks. This not only allows for cleaner and more elegant code, but also helps with debugging as it lets you validate small parts and declare them bug-free.

One big issue that we kept facing was timing. First we did not realize that the ROM had a 2 clock cycle latency, and so we were getting incorrect values from our ROMs. Another timing issue we faced was not properly enabling the read and write lines for the FIFO. In particular, if the FIFO is full, and both read and write enables are raised high, one would expect that the FIFO would shift the new input in while shifting the old output out, remaining full. However, this is not the case. The old output will be shifted out, but the new input will not be shifted in while the FIFO is full, even if it is read on the same clock cycle. Instead, one has to read from the FIFO first, and then write on the following

clock cycle, when the FIFO is not full. Lastly, we encountered a timing issue with our Dot Engine module. We originally attempted to compute the dot product, consisting of 12 16-bit multiplies and 11 32-bit adds, combinationally. We did not realize that this was not possible. When inspecting the outputs from the Dot Engine module with the ILA, we realized that we were not getting correct dot product results. Instead, we had to pipeline the module so that each combinational operation could fit in a single 104MHz clock cycle. We suggest that future groups learn to use the timing report in Vivado in order to spot these issues better than we did.

All in all, we are extremely proud of our end result and had a lot of fun getting there. We accomplished everything we wanted and more, yielding a great-looking project that we can show off to friends. We learned an immense amount about Verilog and how Vivado actually synthesizes and implements Verilog, and we learned valuable skills in regards to project management. We would like to give a tremendous thank you to the 6.111 staff for teaching us, giving us invaluable guidance and debugging help, and giving us the opportunity to succeed.

References

[1] Jonathan Foote. *Automatic Audio Segmentation Using A Measure of Audio Novelty*. In Proc. ICME, volume 1, New York City, New York, USA, 2000.

A Project Verilog

```
parameter BEZIER_BITS = 10;
parameter FRAME_REPEAT_BITS = 1;

parameter SCENE_BITS = 2;
parameter FRAME_BITS = 4;
parameter INSTRUCTION_BITS = 8;

parameter COORD_BITS = 12;
parameter CHROMA_PRECISION = 16;
parameter CHROMA_WIDTH = (12*CHROMA_PRECISION)-1;

parameter MAX_DOT_BITS = 35 - 1;
parameter MAX_DELTA_BITS = 135 - 1;
parameter MAX_TOTAL_BITS = 200 - 1;

// Magic peak number.  Translates to 10^8.
parameter PEAK_THRESHOLD = 200'd1_0000_0000 +
    200'h8000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_0000_00;

```

```
20 // Generate a spectrogram histogram reading from BRAM.
21 module spectro_histogram(
22     input wire clk,
23     input wire [10:0] hcount,
24     input wire [9:0] vcount,
25     input wire blank,
26     input wire [1:0] range,
27     output wire [9:0] vaddr,
28     input wire [15:0] vdata,
29     output reg [2:0] pixel
30     );
31 
32     // 1 bin per pixel, with the selected range
33     assign vaddr = hcount[9:0] >> range;
34 
35     reg [9:0] hheight; // Height of histogram bar
36     reg [9:0] vheight; // The height of pixel above bottom of screen
37     reg blank1; // blank pipelined 1
38 
39     always @(posedge clk) begin
40         // Pipeline stage 1
41         hheight <= vdata >> 7;
42         vheight <= 10'd767 - vcount;
43         blank1 <= blank;
44 
45         // Pipeline stage 2
46         pixel <= blank1 ? 3'b0 : (vheight < hheight) ? 3'b111 :
   3'b0;
47     end
48 
49 endmodule
50 
51 
52 // Generate a chromagram histogram from given chroma.
53 module chroma_histogram(
54     input wire clk,
55     input wire [10:0] hcount,
56     input wire [9:0] vcount,
57     input wire blank,
58     input wire [16*12-1:0] chroma,
59     output reg [2:0] pixel
60     );
61 
62     // 1 bin per pixel, with the selected range
63     wire[3:0] bin_num = hcount[9:0] >> 6;
64 
65     reg [9:0] hheight; // Height of histogram bar
66     reg [9:0] vheight; // The height of pixel above bottom of screen
67     reg blank1; // blank pipelined 1
68 
```

```
69      always @(posedge clk) begin
70          // Pipeline stage 1
71          if (bin_num < 12) hheight <= chroma[bin_num*16 +: 16] >> 6;
72          else hheight <= 0;
73
74          vheight <= 10'd767 - vcount;
75          blank1 <= blank;
76
77          // Pipeline stage 2
78          pixel <= blank1 ? 3'b0 : (vheight < hheight) ? 3'b111 :
    3'b0;
79      end
80
81  endmodule
82
83
84  // Module to bin spectrogram values from the BRAM into a chromagram.
85  // The BROM chroma_bins tells us for every bin in the spectrogram what
86  // bin it belongs to in the chromagram.
87  module chroma_calculator(
88      input wire clk,
89      input wire valid_sample,
90      input wire [11:0] new_sample_addr,
91      input wire [CHROMA_PRECISION-1:0] new_sample_data,
92      input wire last_sample,
93      output reg [CHROMA_WIDTH:0] chroma,
94      output reg done
95  );
96
97      // We are given a spectogram which represents the intensity
98      // of frequencies 0-1024hz. This spectogram is stored in a
99      // Block RAM indexed by spectogram_address.  We seek to iterate through
100     // spectogram indices, and add each value to one of 12 bins.
101
102     parameter INNER_CHROMA_PRECISION = 18;
103     parameter INNER_CHROMA_WIDTH = (INNER_CHROMA_PRECISION*12)-1;
104     parameter BIT_DIFFERENCE = INNER_CHROMA_PRECISION - CHROMA_PRECISION;
105
106     reg[INNER_CHROMA_WIDTH:0] inner_chroma;
107     wire[3:0] chroma_bin;
108     integer i;
109
110     reg [11:0] prev_addr;
111     reg [11:0] prev_prev_addr;
112     reg [11:0] prev_prev_prev_addr;
113     reg [15:0] prev_data;
114     reg [15:0] prev_prev_data;
115     chroma_bins chroma_binz (
116         .clka(clk),     // input wire clka
117         .ena(1),      // input wire ena
```

```
118         .addra(new_sample_addr[10:0]),  // input wire [10 : 0] addra
119         .douta(chroma_bin)  // output wire [3 : 0] douta
120     );
121 
122     always @(posedge clk) begin
123         prev_addr <= new_sample_addr;
124         prev_prev_addr <= prev_addr;
125         prev_prev_prev_addr <= prev_prev_addr;
126         prev_data <= new_sample_data;
127         prev_prev_data <= prev_data;
128 
129         if (valid_sample && (prev_prev_prev_addr != prev_prev_addr)) begin
130             if (last_sample) begin
131                 for (i=0; i<12; i=i+1) begin
132                     chroma[i*CHROMA_PRECISION +: CHROMA_PRECISION]
133                         <= inner_chroma[i*INNER_CHROMA_PRECISION +: INNER_CHROMA_PREC
134                         >> BIT_DIFFERENCE;
135                 end
136 
137                 done <= 1;
138                 inner_chroma <= 0;
139             end
140             else begin
141                 done <= 0;
142 
143                 if (prev_prev_addr < 2048 && chroma_bin < 12) begin
144                     inner_chroma[chroma_bin*INNER_CHROMA_PRECISION +: INNER_CHROMA_PR
145                         <= inner_chroma[chroma_bin*INNER_CHROMA_PRECISION +: INNER_CH
146                 end
147             end
148         end else done <= 0;
149     end
150 
151 endmodule
152 
153 
154 // Heavily pipelined module to perform dot products on two chromagrams.
155 // Has a 4 cycle delay.
156 module dot_engine(
157     input wire clk,
158     input wire[CHROMA_WIDTH:0] dot_a,
159     input wire[CHROMA_WIDTH:0] dot_b,
160     output wire [MAX_DOT_BITS:0] out
161 );
162 
163     reg [31:0] chroma_product_0 = 0;
164     reg [31:0] chroma_product_1 = 0;
165     reg [31:0] chroma_product_2 = 0;
166     reg [31:0] chroma_product_3 = 0;
167     reg [31:0] chroma_product_4 = 0;
```

```
168       reg [31:0] chroma_product_5 = 0;
169       reg [31:0] chroma_product_6 = 0;
170       reg [31:0] chroma_product_7 = 0;
171       reg [31:0] chroma_product_8 = 0;
172       reg [31:0] chroma_product_9 = 0;
173       reg [31:0] chroma_product_10 = 0;
174       reg [31:0] chroma_product_11 = 0;
175
176       reg [32:0] chroma_sum_0 = 0;
177       reg [32:0] chroma_sum_1 = 0;
178       reg [32:0] chroma_sum_2 = 0;
179       reg [32:0] chroma_sum_3 = 0;
180       reg [32:0] chroma_sum_4 = 0;
181       reg [32:0] chroma_sum_5 = 0;
182       reg [33:0] chroma_sum_01 = 0;
183       reg [33:0] chroma_sum_23 = 0;
184       reg [33:0] chroma_sum_45 = 0;
185       reg [33:0] ex_chroma_sum_45 = 0;
186       reg [34:0] chroma_sum_0123 = 0;
187
188
189       always @ (posedge clk) begin
190           chroma_product_0 <=
191               dot_a[1*CHROMA_PRECISION-1:0*CHROMA_PRECISION]
192             * dot_b[1*CHROMA_PRECISION-1:0*CHROMA_PRECISION];
193           chroma_product_1 <=
194               dot_a[2*CHROMA_PRECISION-1:1*CHROMA_PRECISION]
195             * dot_b[2*CHROMA_PRECISION-1:1*CHROMA_PRECISION];
196           chroma_product_2 <=
197               dot_a[3*CHROMA_PRECISION-1:2*CHROMA_PRECISION]
198             * dot_b[3*CHROMA_PRECISION-1:2*CHROMA_PRECISION];
199           chroma_product_3 <=
200               dot_a[4*CHROMA_PRECISION-1:3*CHROMA_PRECISION]
201             * dot_b[4*CHROMA_PRECISION-1:3*CHROMA_PRECISION];
202           chroma_product_4 <=
203               dot_a[5*CHROMA_PRECISION-1:4*CHROMA_PRECISION]
204             * dot_b[5*CHROMA_PRECISION-1:4*CHROMA_PRECISION];
205           chroma_product_5 <=
206               dot_a[6*CHROMA_PRECISION-1:5*CHROMA_PRECISION]
207             * dot_b[6*CHROMA_PRECISION-1:5*CHROMA_PRECISION];
208           chroma_product_6 <=
209               dot_a[7*CHROMA_PRECISION-1:6*CHROMA_PRECISION]
210             * dot_b[7*CHROMA_PRECISION-1:6*CHROMA_PRECISION];
211           chroma_product_7 <=
212               dot_a[8*CHROMA_PRECISION-1:7*CHROMA_PRECISION]
213             * dot_b[8*CHROMA_PRECISION-1:7*CHROMA_PRECISION];
214           chroma_product_8 <=
215               dot_a[9*CHROMA_PRECISION-1:8*CHROMA_PRECISION]
216             * dot_b[9*CHROMA_PRECISION-1:8*CHROMA_PRECISION];
217           chroma_product_9 <=
```

```
218             dot_a[10*CHROMA_PRECISION-1:9*CHROMA_PRECISION]
219           * dot_b[10*CHROMA_PRECISION-1:9*CHROMA_PRECISION];
220         chroma_product_10 <=
221             dot_a[11*CHROMA_PRECISION-1:10*CHROMA_PRECISION]
222           * dot_b[11*CHROMA_PRECISION-1:10*CHROMA_PRECISION];
223         chroma_product_11 <=
224             dot_a[12*CHROMA_PRECISION-1:11*CHROMA_PRECISION]
225           * dot_b[12*CHROMA_PRECISION-1:11*CHROMA_PRECISION];
226 
227         chroma_sum_0 <= chroma_product_0 + chroma_product_1;
228         chroma_sum_1 <= chroma_product_2 + chroma_product_3;
229         chroma_sum_2 <= chroma_product_4 + chroma_product_5;
230         chroma_sum_3 <= chroma_product_6 + chroma_product_7;
231         chroma_sum_4 <= chroma_product_8 + chroma_product_9;
232         chroma_sum_5 <= chroma_product_10 + chroma_product_11;
233         chroma_sum_01 <= chroma_sum_0 + chroma_sum_1;
234         chroma_sum_23 <= chroma_sum_2 + chroma_sum_3;
235         chroma_sum_45 <= chroma_sum_4 + chroma_sum_5;
236         ex_chroma_sum_45 <= chroma_sum_45;
237         chroma_sum_0123 <= chroma_sum_01 + chroma_sum_23;
238     end
239 
240     assign out = chroma_sum_0123 + ex_chroma_sum_45;
241 
242 endmodule
243 
244 
245 // Module to accumulate deltas to structural novelty.
246 module delta_accumulator(
247     input wire clk,
248     input wire rst,
249     input wire new_addition,
250     input wire [MAX_DOT_BITS:0] accumulate,
251     input wire sign,
252     output reg [MAX_DELTA_BITS:0] accumulated
253 );
254 
255     reg state = 0;
256     parameter RESET = 0;
257     parameter COLLECT = 1;
258     always @ (posedge clk) begin
259         case (state)
260             RESET: begin
261                 accumulated <= 1 << MAX_DELTA_BITS;
262                 state <= COLLECT;
263             end
264             COLLECT: begin
265                 if (rst) state <= RESET;
266                 else begin
267                     if (new_addition) begin
```

```
                        if (sign) accumulated <=
                            accumulated - {92'b0, accumulate};
                        else accumulated <=
                            accumulated + {92'b0, accumulate};
                    end
                end
            end
            default: state <= RESET;
        endcase
    end

endmodule


// Module to accumulate total structural novelty.
module total_accumulator(
    input wire clk,
    input wire rst,
    input wire new_addition,
    input wire [MAX_DELTA_BITS:0] accumulate,
    input wire sign,
    output reg [MAX_TOTAL_BITS:0] accumulated
);

    reg state = 0;
    parameter RESET = 0;
    parameter COLLECT = 1;
    always @ (posedge clk) begin
        case (state)
            RESET: begin
                accumulated <= 1 << MAX_TOTAL_BITS;
                state <= COLLECT;
            end
            COLLECT: begin
                if (rst) state <= RESET;
                else begin
                    if (new_addition) begin
                        if (sign) begin
                            if (accumulated > accumulate)
                                accumulated <= accumulated - {65'b0, accumulate};
                            else accumulated <= 0;
                        end
                        else begin
                            if (((1 << (MAX_TOTAL_BITS+1)) - {1'b0, accumulated})
                              > accumulate)
                                accumulated <= accumulated + {65'b0, accumulate};
                            else accumulated <=
                                (1 << (MAX_TOTAL_BITS + 1)) - 1;
                        end
                    end
```

```
                end
            end
            default: state <= RESET;
        endcase
    end

endmodule


// Controller for the chromagram FIFO.
// Can load values, unload, cycle, and shift.
module fifo_controller(
    input wire clk,
    input wire rst,
    input wire [2:0] mode,
    input wire [CHROMA_WIDTH:0] new_fifo_input,
    output wire [CHROMA_WIDTH:0] fifo_output,
    output wire [4:0] data_count,
    output wire fifo_full,
    output wire fifo_empty
);

    reg fifo_read, fifo_write;
    reg [CHROMA_WIDTH:0] fifo_in;
    parameter FIFO_IDLE = 0;
    parameter FIFO_LOAD = 1;
    parameter FIFO_UNLOAD = 2;
    parameter FIFO_CYCLE = 3;
    parameter FIFO_SHIFT = 4;

    always @ (*) begin
        case (mode)
            FIFO_IDLE: begin
                fifo_read = 0;
                fifo_write = 0;
            end
            FIFO_LOAD: begin
                fifo_read = 0;
                fifo_write = 1;
                fifo_in = new_fifo_input;
            end
            FIFO_UNLOAD: begin
                fifo_read = 1;
                fifo_write = 0;
            end
            FIFO_CYCLE: begin
                fifo_read = 1;
                fifo_write = 1;
                fifo_in = fifo_output;
            end
```

```
368             FIFO_SHIFT: begin
369                 fifo_read = 1;
370                 fifo_write = 1;
371                 fifo_in = new_fifo_input;
372             end
373         endcase
374     end
375 
376     chroma_fifo c (
377       .srst(rst),
378       .clk(clk),
379       .din(fifo_in),
380       .wr_en(fifo_write),
381       .rd_en(fifo_read),
382       .dout(fifo_output),
383       .full(fifo_full),
384       .empty(fifo_empty),
385       .data_count(data_count)
386     );
387 
388 
389 endmodule
390 
391 // Controller for the audio buffer FIFO.  Functions identically
392 // to the above.
393 module buffer_controller(
394     input wire clk,
395     input wire rst,
396     input wire [2:0] mode,
397     input wire [11:0] new_fifo_input,
398     output wire [11:0] fifo_output,
399     output wire fifo_full,
400     output wire fifo_empty
401 );
402 
403     reg fifo_read, fifo_write;
404     reg [11:0] fifo_in;
405     parameter FIFO_IDLE = 0;
406     parameter FIFO_LOAD = 1;
407     parameter FIFO_UNLOAD = 2;
408     parameter FIFO_CYCLE = 3;
409     parameter FIFO_SHIFT = 4;
410 
411     always @ (*) begin
412         case (mode)
413             FIFO_IDLE: begin
414                 fifo_read = 0;
415                 fifo_write = 0;
416             end
417             FIFO_LOAD: begin
```

```
                fifo_read = 0;
                fifo_write = 1;
                fifo_in = new_fifo_input;
            end
            FIFO_UNLOAD: begin
                fifo_read = 1;
                fifo_write = 0;
            end
            FIFO_CYCLE: begin
                fifo_read = 1;
                fifo_write = 1;
                fifo_in = fifo_output;
            end
            FIFO_SHIFT: begin
                fifo_read = 1;
                fifo_write = 1;
                fifo_in = new_fifo_input;
            end
        endcase
    end

    buffer_fifo b (
      .srst(rst),
      .clk(clk),
      .din(fifo_in),
      .wr_en(fifo_write),
      .rd_en(fifo_read),
      .dout(fifo_output),
      .full(fifo_full),
      .empty(fifo_empty)
    );


endmodule


// Module to compute structural novelty.  The big boi.
module novelty_calc (
    input wire clk,
    input wire rst,
    input wire [CHROMA_WIDTH:0] new_chroma,
    input wire chroma_done,
    output reg done,
    output reg peak,
);

    parameter FIFO_IDLE = 0;
    parameter FIFO_LOAD = 1;
    parameter FIFO_UNLOAD = 2;
    parameter FIFO_CYCLE = 3;
```

```
468     parameter FIFO_SHIFT = 4;
469
470     parameter NOVELTY_IDLE = 0;
471     parameter NOVELTY_LOAD = 1;
472     parameter NOVELTY_STORE_OLDEST_INT = 2;
473     parameter NOVELTY_STORE_OLDEST = 3;
474     parameter NOVELTY_DOT_NEWEST = 4;
475     parameter NOVELTY_DOT_OLDEST = 5;
476     parameter NOVELTY_COLLECT_MIDDLE = 6;
477     parameter NOVELTY_DOT_MIDDLE = 7;
478     parameter NOVELTY_DOT_MIDDLE_AGAIN = 8;
479     parameter NOVELTY_FINISH_DOTTING_MIDDLE = 9;
480     parameter NOVELTY_ADD_DELTA = 10;
481     parameter NOVELTY_MOVING_AVERAGE = 11;
482     parameter NOVELTY_PEAK = 12;
483
484     reg [3:0] state;
485     reg [2:0] fifo_mode;
486     reg [CHROMA_WIDTH:0] new_fifo_input;
487     wire [CHROMA_WIDTH:0] fifo_output;
488     wire [4:0] data_count;
489     wire fifo_full, fifo_empty;
490
491     reg [4:0] current_index;
492     reg [4:0] current_dot_index;
493
494     reg [CHROMA_WIDTH:0] dot_a, dot_b;
495     wire [MAX_DOT_BITS:0] dot_out;
496
497     reg add_to_total_novelty, add_to_delta_novelty;
498     reg add_to_total_novelty_sign, add_to_delta_novelty_sign;
499     wire [MAX_TOTAL_BITS:0] accumulated_total_novelty;
500     wire [MAX_DELTA_BITS:0] accumulated_delta_novelty;
501     reg [MAX_DELTA_BITS:0] total_novelty_addition;
502     reg delta_novelty_reset;
503
504     reg [CHROMA_WIDTH:0] oldest_chroma, newest_chroma, middle_chroma;
505
506     reg [MAX_TOTAL_BITS:0] old_total_novelty;
507     reg [MAX_TOTAL_BITS:0] mid_total_novelty;
508     reg [MAX_TOTAL_BITS:0] average_total_novelty;
509
510     reg [4:0] current_index_m1 = 0;
511     reg [4:0] current_index_m2 = 0;
512     reg [4:0] current_index_m3 = 0;
513     reg [4:0] current_index_m4 = 0;
514
515     // Clocked block to define state transitions.
516     always @ (posedge clk) begin
517         if (fifo_mode == FIFO_UNLOAD || fifo_mode == FIFO_CYCLE)
```

```
518             current_index <= current_index - 1;
519
520         current_index_m1 <= current_index;
521         current_index_m2 <= current_index_m1;
522         current_index_m3 <= current_index_m2;
523         current_index_m4 <= current_index_m3;
524         case (state)
525             NOVELTY_IDLE: begin
526
527                 oldest_chroma <= 0;
528                 middle_chroma <= 0;
529                 current_index <= 0;
530                 peak <= 0;
531                 done <= 0;
532
533                 if (chroma_done) begin
534                     newest_chroma <= new_chroma;
535                     if (fifo_full) begin
536                         state <= NOVELTY_STORE_OLDEST_INT;
537
538                     end
539                     else begin
540                         state <= NOVELTY_LOAD;
541                     end
542                 end
543                 else newest_chroma <= 0;
544             end
545
546             NOVELTY_LOAD: begin
547                 state <= NOVELTY_IDLE;
548             end
549
550             NOVELTY_STORE_OLDEST_INT: begin
551                 state <= NOVELTY_STORE_OLDEST;
552             end
553
554             NOVELTY_STORE_OLDEST: begin
555                 oldest_chroma <= fifo_output;
556                 state <= NOVELTY_DOT_NEWEST;
557             end
558
559             NOVELTY_DOT_NEWEST: begin
560                 if (current_index == 0)
561                     state <= NOVELTY_DOT_OLDEST;
562             end
563
564             NOVELTY_DOT_OLDEST: begin
565                 if (current_index == 0)
566                     state <= NOVELTY_COLLECT_MIDDLE;
567             end
```

```
568
569             NOVELTY_COLLECT_MIDDLE: begin
570                 if (current_index == 16)
571                     middle_chroma <= fifo_output;
572                 else if (current_index == 0)
573                     state <= NOVELTY_DOT_MIDDLE;
574             end
575
576             NOVELTY_DOT_MIDDLE: begin
577                 if (current_index == 0)
578                     state <= NOVELTY_DOT_MIDDLE_AGAIN;
579             end
580
581             NOVELTY_DOT_MIDDLE_AGAIN: begin
582                 if (current_index == 1)
583                     state <= NOVELTY_FINISH_DOTTING_MIDDLE;
584             end
585
586             NOVELTY_FINISH_DOTTING_MIDDLE: begin
587                 if (current_index_m4 == 1)
588                     state <= NOVELTY_ADD_DELTA;
589             end
590
591             NOVELTY_ADD_DELTA: begin
592                 state <= NOVELTY_MOVING_AVERAGE;
593             end
594
595             NOVELTY_MOVING_AVERAGE: begin
596                 average_total_novelty <= (
597                     {1'b0, average_total_novelty}
598                   + {1'b0, accumulated_total_novelty}
599                 ) >> 1;
600
601                 state <= NOVELTY_PEAK;
602             end
603
604             NOVELTY_PEAK: begin
605                 done <= 1;
606                 peak <= (mid_total_novelty > average_total_novelty)
607                      && (mid_total_novelty > old_total_novelty)
608                      && (mid_total_novelty > PEAK_THRESHOLD);
609
610                 state <= NOVELTY_IDLE;
611                 mid_total_novelty <= average_total_novelty;
612                 old_total_novelty <= mid_total_novelty;
613             end
614
615         endcase
616
617     end
```

```
618
619     // Combinational assignment of control signals to fifo controller
620     // and accumulators based off the current state.
621     always @ (*) begin
622         case (state)
623
624             NOVELTY_IDLE: begin
625                 fifo_mode = FIFO_IDLE;
626                 new_fifo_input = 0;
627                 add_to_total_novelty = 0;
628                 add_to_delta_novelty = 0;
629                 add_to_total_novelty_sign = 0;
630                 add_to_delta_novelty_sign = 0;
631                 dot_a = 0;
632                 dot_b = 0;
633                 delta_novelty_reset = 0;
634             end
635
636             NOVELTY_LOAD: begin
637                 fifo_mode = FIFO_LOAD;
638                 new_fifo_input = newest_chroma;
639                 add_to_total_novelty = 0;
640                 add_to_delta_novelty = 0;
641                 add_to_total_novelty_sign = 0;
642                 add_to_delta_novelty_sign = 0;
643                 dot_a = 0;
644                 dot_b = 0;
645                 delta_novelty_reset = 1;
646             end
647
648             NOVELTY_STORE_OLDEST_INT: begin
649                 fifo_mode = FIFO_UNLOAD;
650                 new_fifo_input = 0;
651                 add_to_total_novelty = 0;
652                 add_to_delta_novelty = 0;
653                 add_to_total_novelty_sign = 0;
654                 add_to_delta_novelty_sign = 0;
655                 dot_a = 0;
656                 dot_b = 0;
657                 delta_novelty_reset = 0;
658             end
659
660             NOVELTY_STORE_OLDEST: begin
661                 fifo_mode = FIFO_SHIFT;
662                 new_fifo_input = newest_chroma;
663                 add_to_total_novelty = 0;
664                 add_to_delta_novelty = 0;
665                 add_to_total_novelty_sign = 0;
666                 add_to_delta_novelty_sign = 0;
667                 dot_a = 0;
```

```
668                 dot_b = 0;
669                 delta_novelty_reset = 0;
670             end
671
672             NOVELTY_DOT_NEWEST: begin
673                 fifo_mode = FIFO_CYCLE;
674                 new_fifo_input = 0;
675                 add_to_total_novelty = 0;
676                 add_to_delta_novelty = current_index_m4 != 0;
677                 add_to_total_novelty_sign = 0;
678                 add_to_delta_novelty_sign = current_index_m4 > 15;
679                 dot_a = newest_chroma;
680                 dot_b = fifo_output;
681                 delta_novelty_reset = 0;
682             end
683
684             NOVELTY_DOT_OLDEST: begin
685                 fifo_mode = FIFO_CYCLE;
686                 new_fifo_input = 0;
687                 add_to_total_novelty = 0;
688                 add_to_delta_novelty = current_index_m4 != 0;
689                 add_to_total_novelty_sign = 0;
690                 add_to_delta_novelty_sign = current_index_m4 > 16;
691                 dot_a = oldest_chroma;
692                 dot_b = fifo_output;
693                 delta_novelty_reset = 0;
694             end
695
696             NOVELTY_COLLECT_MIDDLE: begin
697                 fifo_mode = FIFO_CYCLE;
698                 new_fifo_input = 0;
699                 add_to_total_novelty = 0;
700                 add_to_delta_novelty = (current_index_m4 < 4)
701                                     && (current_index_m4 != 0);
702                 add_to_total_novelty_sign = 0;
703                 add_to_delta_novelty_sign = 0;
704                 dot_a = 0;
705                 dot_b = 0;
706                 delta_novelty_reset = 0;
707             end
708
709             NOVELTY_DOT_MIDDLE: begin
710                 fifo_mode = FIFO_CYCLE;
711                 new_fifo_input = 0;
712                 add_to_total_novelty = 0;
713                 add_to_delta_novelty = (current_index_m4 != 16)
714                                     && (current_index_m4 != 0);
715                 add_to_total_novelty_sign = 0;
716                 add_to_delta_novelty_sign = current_index_m4 < 16;
717                 dot_a = middle_chroma;
```

```
718                 dot_b = fifo_output;
719                 delta_novelty_reset = 0;
720             end
721
722             NOVELTY_DOT_MIDDLE_AGAIN: begin
723                 fifo_mode = FIFO_CYCLE;
724                 new_fifo_input = 0;
725                 add_to_total_novelty = 0;
726                 add_to_delta_novelty = (current_index_m4 != 16)
727                                     && (current_index_m4 != 0);
728                 add_to_total_novelty_sign = 0;
729                 add_to_delta_novelty_sign = current_index_m4 < 16;
730                 dot_a = middle_chroma;
731                 dot_b = fifo_output;
732                 delta_novelty_reset = 0;
733             end
734
735             NOVELTY_FINISH_DOTTING_MIDDLE: begin
736                 fifo_mode = (current_index_m4 == 4)
737                           ? FIFO_LOAD : FIFO_IDLE;
738                 new_fifo_input = newest_chroma;
739                 add_to_total_novelty = 0;
740                 add_to_delta_novelty = 1;
741                 add_to_total_novelty_sign = 0;
742                 add_to_delta_novelty_sign = 1;
743                 dot_a = 0;
744                 dot_b = 0;
745                 delta_novelty_reset = 0;
746             end
747
748             NOVELTY_ADD_DELTA: begin
749                 fifo_mode = FIFO_IDLE;
750                 new_fifo_input = 0;
751                 add_to_total_novelty = 1;
752                 add_to_delta_novelty = 0;
753                 add_to_total_novelty_sign = accumulated_delta_novelty
754                                           < (1 << MAX_DELTA_BITS);
755                 add_to_delta_novelty_sign = 0;
756                 dot_a = 0;
757                 dot_b = 0;
758                 delta_novelty_reset = 1;
759                 total_novelty_addition = (accumulated_delta_novelty
760                                         < (1 << MAX_DELTA_BITS))
761                                         ? ((1 << MAX_DELTA_BITS) -
762                                            accumulated_delta_novelty)
763                                         : (accumulated_delta_novelty -
764                                           (1 << MAX_DELTA_BITS));
765             end
766
767             NOVELTY_MOVING_AVERAGE: begin
```

```
768                 fifo_mode = FIFO_IDLE;
769                 new_fifo_input = 0;
770                 add_to_total_novelty = 0;
771                 add_to_delta_novelty = 0;
772                 add_to_total_novelty_sign = 0;
773                 add_to_delta_novelty_sign = 0;
774                 dot_a = 0;
775                 dot_b = 0;
776                 delta_novelty_reset = 0;
777                 total_novelty_addition = 0;
778             end
779 
780             NOVELTY_PEAK: begin
781                 fifo_mode = FIFO_IDLE;
782                 new_fifo_input = 0;
783                 add_to_total_novelty = 0;
784                 add_to_delta_novelty = 0;
785                 add_to_total_novelty_sign = 0;
786                 add_to_delta_novelty_sign = 0;
787                 dot_a = 0;
788                 dot_b = 0;
789                 delta_novelty_reset = 0;
790                 total_novelty_addition = 0;
791             end
792         endcase
793     end
794 
795     dot_engine dotter(
796         .clk(clk),
797         .dot_a(dot_a),
798         .dot_b(dot_b),
799         .out(dot_out)
800     );
801 
802     fifo_controller fifo_control(
803         .clk(clk),
804         .rst(rst),
805         .mode(fifo_mode),
806         .new_fifo_input(new_fifo_input),
807         .fifo_output(fifo_output),
808         .data_count(data_count),
809         .fifo_full(fifo_full),
810         .fifo_empty(fifo_empty)
811     );
812 
813     total_accumulator total_novelty(
814         .clk(clk),
815         .rst(rst),
816         .new_addition(add_to_total_novelty),
817         .accumulate(total_novelty_addition),
```

```
818         .sign(add_to_total_novelty_sign),
819         .accumulated(accumulated_total_novelty)
820     );
821 
822     delta_accumulator delta_novelty(
823         .clk(clk),
824         .rst(delta_novelty_reset),
825         .new_addition(add_to_delta_novelty),
826         .accumulate(dot_out),
827         .sign(add_to_delta_novelty_sign),
828         .accumulated(accumulated_delta_novelty)
829     );
830 
831 endmodule
832 
833 // Nice helper function to flip the bits in a wire.
834 // Found on stackexchange, no idea how it works but it does.
835 function[11:0] bit_order (input[11:0] data);
836 integer i;
837 begin
838     for (i=0; i < 12; i=i+1) begin : reverse
839         bit_order[11-i] = data[i];
840     end
841 end
842 endfunction
843 
844 
845 // Combinational module to compute the current position
846 // on a Bezier curve.
847 module bezier_coordinate(
848     input wire[COORD_BITS-1:0] p0,
849     input wire[COORD_BITS-1:0] p1,
850     input wire[COORD_BITS-1:0] p2,
851     input wire[COORD_BITS-1:0] p3,
852     input wire[BEZIER_BITS-1:0] t,
853     output wire[COORD_BITS-1:0] out
854 );
855 
856     // Full equation:
857     // (1-t)^3 * p0 + 3*t*(1-t)^2 * p1 + 3*t^2*(1-t)*p2 + t^3*p3
858 
859     wire[50:0] t_com   = (2**BEZIER_BITS-1) - t;
860     wire[50:0] t_com_2 = ({32'd0,t_com}*t_com) >> BEZIER_BITS;
861     wire[50:0] t_com_3 = ({32'd0,t_com}*t_com_2) >> BEZIER_BITS;
862     wire[50:0] t_2     = ({32'd0,t}*t) >> BEZIER_BITS;
863     wire[50:0] t_3     = ({32'd0,t}*t_2) >> BEZIER_BITS;
864 
865     assign out = (t_com_3 * p0
866                 + ((3 * t * t_com_2 * p1)>>>BEZIER_BITS)
867                 + ((3 * t_2 * t_com * p2)>>>BEZIER_BITS)
```

```
                + t_3 * p3) >>> BEZIER_BITS;
endmodule


// Interpolator module reads Instruction ROM, and interpolates
// between bezier curves in the given scene.
module interpolator(
    input wire clk,
    input wire[SCENE_BITS-1:0] scene,

    output wire[COORD_BITS-1:0] x,
    output wire[COORD_BITS-1:0] y,
    output wire laser_on
);

    reg[FRAME_BITS-1:0] frame;
    reg[INSTRUCTION_BITS-1:0] instruction;

    wire[SCENE_BITS+FRAME_BITS+INSTRUCTION_BITS-1:0] address = {
        scene, frame, instruction
    };

    wire[8*COORD_BITS:0] instruction_out;

    wire[COORD_BITS-1:0] p0x =
        instruction_out[8*COORD_BITS:7*COORD_BITS+1];
    wire[COORD_BITS-1:0] p0y =
        instruction_out[7*COORD_BITS:6*COORD_BITS+1];
    wire[COORD_BITS-1:0] p1x =
        instruction_out[6*COORD_BITS:5*COORD_BITS+1];
    wire[COORD_BITS-1:0] p1y =
        instruction_out[5*COORD_BITS:4*COORD_BITS+1];
    wire[COORD_BITS-1:0] p2x =
        instruction_out[4*COORD_BITS:3*COORD_BITS+1];
    wire[COORD_BITS-1:0] p2y =
        instruction_out[3*COORD_BITS:2*COORD_BITS+1];
    wire[COORD_BITS-1:0] p3x =
        instruction_out[2*COORD_BITS:COORD_BITS+1];
    wire[COORD_BITS-1:0] p3y =
        instruction_out[COORD_BITS:1];
    assign laser_on          = instruction_out[0];

    reg[BEZIER_BITS-1:0] time_in_inst = 0;
    reg[FRAME_REPEAT_BITS-1:0] time_in_frame = 0;
    reg[SCENE_BITS-1:0] last_scene;

    instruction_rom rom (
       .clka(clk),
       .ena(1),
       .addra(address),
```

```
        .douta(instruction_out)
    );

    bezier_coordinate bx (
        .p0(p0x),
        .p1(p1x),
        .p2(p2x),
        .p3(p3x),
        .t(time_in_inst),
        .out(x)
    );

    bezier_coordinate by (
        .p0(p0y),
        .p1(p1y),
        .p2(p2y),
        .p3(p3y),
        .t(time_in_inst),
        .out(y)
    );

    always @(posedge clk) begin
        last_scene <= scene;

        // When the scene changes, reset variables
        if (last_scene != scene) begin
            time_in_frame <= 0;
            time_in_inst <= 0;
            frame <= 0;
            instruction <= 0;
        end else begin
            time_in_inst <= time_in_inst + 1;

            // Increment the instruction before an overflow
            if (&time_in_inst) begin
                instruction <= instruction + 1;

                // Increment the frame if not repeating
                if (&instruction) begin
                    time_in_frame <= time_in_frame + 1;

                    // Increment the frame before overflow
                    if (&time_in_frame) frame <= frame + 1;
                end
            end
        end
    end

endmodule

```

```
// SPI module for MCP4822 DAC using both channels.
// Alternates between the X and Y channel, and thus
// updates each channel half as fast as audio_SPI.
module laser_SPI(
    input wire clock,
    input wire [11:0] x,
    input wire [11:0] y,
    output reg cs,
    output wire s_out // mosi
    );

    reg[1:0] state;
    reg[15:0] instruction;
    reg[6:0] instruction_sent;
    reg channel;

    parameter IDLE = 0;
    parameter SEND = 1;

    assign s_out = instruction[0];

    always @(posedge clock) begin
        case (state)
            IDLE: begin
                instruction[0] <= channel;
                instruction[3:1] <= 3'b100;
                instruction[15:4] <= channel
                                     ? bit_order(x) : bit_order(y);
                instruction_sent <= 0;
                state <= SEND;
                cs <= 0;
            end

            SEND: begin
                instruction[14:0] <= instruction[15:1];
                instruction_sent <= instruction_sent + 1;

                if (instruction_sent >= 16) begin
                    cs <= 1;
                    channel = ~channel;
                    state <= IDLE;
                end
            end
            default: state <= IDLE;
        endcase
    end

endmodule
```

```
1018 // Similar to laser_SPI, but only outputs on one channel at twice
1019 // the frequency and half the gain.
1020 module audio_SPI(
1021     input wire clock,
1022     input wire [11:0] audio,
1023     output reg cs,
1024     output wire s_out // mosi
1025     );
1026 
1027     reg[1:0] state;
1028     reg[15:0] instruction;
1029     reg[6:0] instruction_sent;
1030 
1031     parameter IDLE = 0;
1032     parameter SEND = 1;
1033 
1034     assign s_out = instruction[0];
1035 
1036     always @(posedge clock) begin
1037         case (state)
1038             IDLE: begin
1039                 instruction[3:0] <= 4'b1100;
1040                 instruction[15:4] <= bit_order(audio);
1041                 instruction_sent <= 0;
1042                 state <= SEND;
1043                 cs <= 0;
1044             end
1045 
1046             SEND: begin
1047                 instruction[14:0] <= instruction[15:1];
1048                 instruction_sent <= instruction_sent + 1;
1049 
1050                 if (instruction_sent >= 16) begin
1051                     cs <= 1;
1052                     state <= IDLE;
1053                 end
1054             end
1055             default: state <= IDLE;
1056         endcase
1057     end
1058 
1059 endmodule
1060 
1061 
1062 // Main module.  Wires up all the major components.
1063 module main (
1064     input wire CLK100MHZ,
1065     input wire [15:0] SW,
1066     input wire BTNC, BTNU, BTNL, BTNR, BTND,
1067     input wire AD3P, AD3N,  // The top pair of ports on JXADC on Nexys 4
```

```
1068        output wire [3:0] VGA_R,
1069        output wire [3:0] VGA_B,
1070        output wire [3:0] VGA_G,
1071        output wire VGA_HS,
1072        output wire VGA_VS,
1073        output wire AUD_PWM, AUD_SD,
1074        output wire LED16_B, LED16_G, LED16_R,
1075        output wire LED17_B, LED17_G, LED17_R,
1076        output wire [15:0] LED, // LEDs above switches
1077        output wire [7:0] SEG,  // segments A-G (0-6), DP (7)
1078        output wire [7:0] AN,    // Display 0-7
1079        output wire [7:0] JB
1080        );
1081
1082        // SETUP CLOCKS
1083        // 104Mhz clock for XADC and primary clock domain
1084        // It divides by 4 and runs the ADC clock at 26Mhz
1085        // And the ADC can do one conversion in 26 clock cycles
1086        // So the sample rate is 1Msps (not posssible w/ 100Mhz)
1087        // 65Mhz for VGA Video
1088        // 208mhz for ILA
1089        // 15mhz for DAC SPI
1090        // 6mhz for graphics generation
1091        wire clk_104mhz, clk_65mhz, clk_208mhz;
1092        clk_wiz_0 clockgen(
1093            .clk_in1(CLK100MHZ),
1094            .clk_out1(clk_104mhz),
1095            .clk_out2(clk_65mhz),
1096            .clk_out3(clk_208mhz),
1097            .clk_out4(JB[5]),
1098            .clk_out5(JB[1])
1099        );
1100
1101        // INSTANTIATE XVGA SIGNALS (1024x768)
1102        wire [10:0] hcount;
1103        wire [9:0] vcount;
1104        wire hsync, vsync, blank;
1105        xvga xvga1(
1106            .vclock(clk_65mhz),
1107            .hcount(hcount),
1108            .vcount(vcount),
1109            .vsync(vsync),
1110            .hsync(hsync),
1111            .blank(blank));
1112
1113
1114
1115        // Initiate 7seg display to show novelty value.
1116        wire[31:0] display_novelty;
1117        display_8hex display(
```

```
1118            .clk(clk_65mhz),
1119            .data(display_novelty),
1120            .seg(SEG[6:0]),
1121            .strobe(AN));
1122        assign SEG[7] = 1;
1123
1124        // Parametrized debounce module to do all 16 switches and 5 buttons
1125        wire BTNC_clean, BTNU_clean, BTND_clean, BTNL_clean, BTNR_clean;
1126        wire [15:0] SW_clean;
1127        debounce #(.COUNT(21)) db0 (
1128            .clk(clk_104mhz),
1129            .reset(1'b0),
1130            .noisy({SW, BTNC, BTNU, BTND, BTNL, BTNR}),
1131            .clean({SW_clean, BTNC_clean, BTNU_clean,
1132                    BTND_clean, BTNL_clean, BTNR_clean})
1133        );
1134
1135
1136        // Initiate XADC IP.
1137        wire [15:0] sample_reg;
1138        wire eoc, xadc_reset;
1139        xadc_demo xadc_demo (
1140            .dclk_in(clk_104mhz),
1141            .di_in(0),
1142            .daddr_in(6'h13),     r
1143            .den_in(1),
1144            .dwe_in(0),
1145            .drdy_out(),
1146            .do_out(sample_reg),
1147            .reset_in(xadc_reset),
1148            .vp_in(0),
1149            .vn_in(0),
1150            .vauxp3(AD3P),
1151            .vauxn3(AD3N),
1152            .channel_out(),
1153            .eoc_out(eoc),
1154            .alarm_out(),
1155            .eos_out(),
1156            .busy_out()
1157        );
1158        assign xadc_reset = BTNC_clean;
1159
1160        // INSTANTIATE 64x OVERSAMPLING
1161        // This outputs 15-bit samples at a 62.5/4kHz sample rate
1162        // (3 more bits, 1/64 the sample rate)
1163        wire [14:0] osample64;
1164        reg [14:0] prev_osample64;
1165        reg [14:0] prev_prev_osample64;
1166        wire done_osample64;
1167        reg prev_done_osample64;
```

```
1168        reg prev_prev_done_osample64;
1169        oversample64 osamp64_1 (
1170            .clk(clk_104mhz),
1171            .sample(sample_reg[15:4]),
1172            .eoc(eoc),
1173            .oversample(osample64),
1174            .done(done_osample64));
1175
1176        always @ (posedge clk_104mhz) begin
1177            prev_osample64 <= osample64;
1178            prev_prev_osample64 <= prev_osample64;
1179            prev_done_osample64 <= done_osample64;
1180            prev_prev_done_osample64 <= prev_done_osample64;
1181        end
1182
1183
1184        // Audio buffer FIFO for ~4 second audio delay.
1185        parameter FIFO_IDLE = 0;
1186        parameter FIFO_LOAD = 1;
1187        parameter FIFO_UNLOAD = 2;
1188        parameter FIFO_CYCLE = 3;
1189        parameter FIFO_SHIFT = 4;
1190
1191        reg [3:0] buffer_mode;
1192        reg [11:0] new_buffer_input;
1193        wire [11:0] buffer_output;
1194        wire buffer_full, buffer_empty;
1195        buffer_controller buffer(
1196            .clk(clk_104mhz),
1197            .rst(0),
1198            .mode(buffer_mode),
1199            .new_fifo_input(new_buffer_input),
1200            .fifo_output(buffer_output),
1201            .fifo_full(buffer_full),
1202            .fifo_empty(buffer_empty)
1203        );
1204
1205        parameter BUFFER_IDLE = 0;
1206        parameter BUFFER_LOAD = 1;
1207        parameter BUFFER_UNLOAD = 2;
1208        reg [1:0] buffer_state = BUFFER_IDLE;
1209        always @ (posedge clk_104mhz) begin
1210            case (buffer_state)
1211                BUFFER_IDLE: begin
1212                    if (prev_prev_done_osample64) begin
1213                        new_buffer_input <= prev_prev_osample64 >> 3;
1214                        if (buffer_full) begin
1215                            buffer_state <= BUFFER_UNLOAD;
1216                        end
1217                        else begin
```

```
                        buffer_state <= BUFFER_LOAD;
                    end
                end
            end
            BUFFER_LOAD: begin
                buffer_state <= BUFFER_IDLE;
            end
            BUFFER_UNLOAD: begin
                buffer_state <= BUFFER_LOAD;
            end
        endcase
    end

    always @ (*) begin
        case (buffer_state)
            BUFFER_IDLE: begin
                buffer_mode = FIFO_IDLE;
            end
            BUFFER_LOAD: begin
                buffer_mode = FIFO_LOAD;
            end
            BUFFER_UNLOAD: begin
                buffer_mode = FIFO_UNLOAD;
            end
        endcase
    end

    // Output audio FIFO output to DAC over SPI.
    audio_SPI s(
        .clock(JB[5]),
        .audio(buffer_output),
        .cs(JB[6]),
        .s_out(JB[4])
    );

    // Instantiate audio sample block RAM, which stores
    // 4096 16 bit audio samples.
    wire fwe;
    reg [11:0] fhead = 0;
    reg [11:0] prev_fhead;
    reg [11:0] prev_prev_fhead;
    wire [15:0] fdata, fsample, fsample_regular, fsample_hanning;
    wire [11:0] faddr;
    bram_frame bram1 (
        .clka(clk_104mhz),
        .wea(fwe),
        .addra(prev_prev_fhead),
        .dina(fsample),
        .clkb(clk_104mhz),
        .addrb(faddr),
```

```
1268            .doutb(fdata)
1269        );
1270
1271        // Instatiate BROM for hanning window coefficients.
1272        wire [15:0] hanning_value;
1273        hanning hanning_values(
1274            .clka(clk_104mhz),
1275            .ena(1),
1276            .addra(fhead),
1277            .douta(hanning_value)
1278        );
1279
1280        // SAMPLE FRAME BRAM WRITE PORT SETUP
1281        always @(posedge clk_104mhz) begin
1282            // Move the pointer every oversample
1283            if (done_osample64) fhead <= fhead + 1;
1284            prev_fhead <= fhead;
1285            prev_prev_fhead <= prev_fhead;
1286        end
1287
1288        // Pad the oversample with zeros to pretend it's 16 bits
1289        assign fsample_hanning = ({16'b0, prev_prev_osample64, 1'b0}
1290                                  * {16'b0, hanning_value}) >> 16;
1291        assign fsample_regular = {prev_prev_osample64, 1'b0};
1292        assign fsample = SW_clean[3] ? fsample_hanning : fsample_regular;
1293
1294        // Write only when we finish an oversample
1295        // (every 104*16 clock cycles)
1296        assign fwe = prev_prev_done_osample64;
1297
1298        // SAMPLE FRAME BRAM READ PORT SETUP
1299        wire vsync_104mhz, vsync_104mhz_pulse;
1300        synchronize vsync_synchronize(
1301            .clk(clk_104mhz),
1302            .in(vsync),
1303            .out(vsync_104mhz));
1304
1305        level_to_pulse vsync_ltp(
1306            .clk(clk_104mhz),
1307            .level(~vsync_104mhz),
1308            .pulse(vsync_104mhz_pulse));
1309
1310        // INSTANTIATE BRAM TO FFT MODULE
1311        // This module handles the magic of reading sample frames
1312        // from the BRAM whenever start is asserted, and sending
1313        // it to the FFT block design over the AXI-stream interface.
1314        wire collected_frame;
1315        assign collected_frame = prev_prev_done_osample64
1316                                 && (& prev_prev_fhead);
1317
```

```
1318        // All these are control lines to the FFT block design
1319        wire last_missing;
1320        wire [31:0] frame_tdata;
1321        wire frame_tlast, frame_tready, frame_tvalid;
1322        bram_to_fft bram_to_fft_0(
1323            .clk(clk_104mhz),
1324            .head(prev_prev_fhead),
1325            .addr(faddr),
1326            .data(fdata),
1327            .start(collected_frame),
1328            .last_missing(last_missing),
1329            .frame_tdata(frame_tdata),
1330            .frame_tlast(frame_tlast),
1331            .frame_tready(frame_tready),
1332            .frame_tvalid(frame_tvalid)
1333        );
1334
1335        // FFT module, implemented as a block design with
1336        // a 4096pt, 16bit FFT that outputs in magnitude by
1337        // doing sqrt(Re^2 + Im^2) on the FFT result.
1338        //
1339        // It's fully pipelined, so it streams 4096-wide frames
1340        // of frequency data as fast as you stream in 4096-wide
1341        // frames of time-domain samples.
1342
1343        // FFT magnitude for the current index
1344        wire [23:0] magnitude_tdata;
1345
1346        // Current index being output, from 0 to 4096
1347        wire [11:0] magnitude_tuser;
1348
1349        // Adjusts the scaling of the FFT
1350        wire [11:0] scale_factor;
1351        wire magnitude_tlast, magnitude_tvalid;
1352        fft_mag fft_mag_i(
1353            .clk(clk_104mhz),
1354            .event_tlast_missing(last_missing),
1355            .frame_tdata(frame_tdata),
1356            .frame_tlast(frame_tlast),
1357            .frame_tready(frame_tready),
1358            .frame_tvalid(frame_tvalid),
1359            .scaling(SW_clean[15:4]),
1360            .magnitude_tdata(magnitude_tdata),
1361            .magnitude_tlast(magnitude_tlast),
1362            .magnitude_tuser(magnitude_tuser),
1363            .magnitude_tvalid(magnitude_tvalid));
1364
1365        // Only care about the range from index 0 to 1023,
1366        // which represents frequencies 0 to omega/2
1367        // where omega is the nyquist frequency (sample rate / 2)
```

```
1368        wire in_range = ~|magnitude_tuser[11:10];
1369
1370        // Instantiate 16x1024 bram for storing histogram data.
1371        wire [9:0] haddr; // The read port address
1372        wire [15:0] hdata; // The read port data
1373        bram_fft bram2 (
1374            .clka(clk_104mhz),
1375            // Only save if in range and valid
1376            .wea(in_range & magnitude_tvalid),
1377            .addra(magnitude_tuser[9:0]),
1378            .dina(magnitude_tdata[15:0]),
1379            .clkb(clk_65mhz),
1380            .addrb(haddr),
1381            .doutb(hdata)
1382        );
1383
1384
1385        // Histogram generation modules for chromagram and
1386        // spectrogram, switchable by switch 2.
1387        wire [2:0] hist_pixel;
1388        wire [2:0] chroma_pixel;
1389        wire [2:0] spec_pixel;
1390        wire [1:0] hist_range;
1391        wire[12*16-1:0] chroma;
1392
1393        chroma_histogram chroma_histogram(
1394            .clk(clk_65mhz),
1395            .hcount(hcount),
1396            .vcount(vcount),
1397            .blank(blank),
1398            .chroma(chroma),
1399            .pixel(chroma_pixel)
1400        );
1401
1402        spectro_histogram histo(
1403            .clk(clk_65mhz),
1404            .hcount(hcount),
1405            .vcount(vcount),
1406            .blank(blank),
1407            .range(SW_clean[1:0]),
1408            .vaddr(haddr),
1409            .vdata(hdata),
1410            .pixel(spec_pixel)
1411        );
1412
1413        // Switch display to chroma or spectro.
1414        assign hist_pixel = SW_clean[2] ? chroma_pixel : spec_pixel;
1415
1416        // Chroma calculation hookup.
1417        wire chroma_done;
```

```
1418      chroma_calculator chroma_calc1(
1419          .clk(clk_104mhz),
1420          .valid_sample(magnitude_tvalid),
1421          .new_sample_addr(magnitude_tuser),
1422          .new_sample_data(magnitude_tdata[15:0]),
1423          .last_sample(magnitude_tlast),
1424          .chroma(chroma),
1425          .done(chroma_done)
1426      );
1427
1428      // Structual novelty calculator module hookup.
1429      wire peak_done, peak;
1430      novelty_calc nov(
1431          .clk(clk_104mhz),
1432          .rst(BTNC_clean),
1433          .new_chroma(chroma),
1434          .chroma_done(chroma_done),
1435          .done(peak_done),
1436          .peak(peak)
1437      );
1438
1439      // Flip LED and change scene on keak.
1440      reg led_reg = 0;
1441      reg [SCENE_BITS-1:0] scene;
1442
1443      always @ (posedge clk_104mhz) begin
1444          if (peak && peak_done) begin
1445              led_reg <= ~ led_reg;
1446              scene <= scene + 1;
1447          end
1448      end assign LED16_B = led_reg;
1449
1450
1451      // Interpolator and galvo DAC output.
1452      wire [11:0] x, y;
1453      wire laser_on;
1454
1455      laser_SPI sc(.clock(JB[1]),
1456             .x(x),
1457             .y(y),
1458             .cs(JB[2]),
1459             .s_out(JB[0])
1460      );
1461
1462     interpolator i(
1463         .clk(JB[1]),
1464         .scene(scene),
1465         .x(x),
1466         .y(y),
1467         .laser_on(laser_on)
```

```
1468     );
1469
1470     assign JB[3] = !laser_on;
1471
1472
1473      // VGA OUTPUT
1474      // Histogram has two pipeline stages so we'll
1475      // pipeline the hs and vs accordingly
1476      reg [1:0] hsync_delay;
1477      reg [1:0] vsync_delay;
1478      reg hsync_out, vsync_out;
1479      always @(posedge clk_65mhz) begin
1480          {hsync_out,hsync_delay} <= {hsync_delay,hsync};
1481          {vsync_out,vsync_delay} <= {vsync_delay,vsync};
1482      end
1483      assign VGA_R = {4{hist_pixel[0]}};
1484      assign VGA_G = {4{hist_pixel[1]}};
1485      assign VGA_B = {4{hist_pixel[2]}};
1486      assign VGA_HS = hsync_out;
1487      assign VGA_VS = vsync_out;
1488
1489
1490 endmodule
1491
1492 // Below are helper modules written by 6.111
1493 // course staff or Mitchell Gu.
1494
1495 // BRAM to FFT interfacer, written by Mitchell Gu.
1496 module bram_to_fft(
1497     input wire clk,
1498     input wire [11:0] head,
1499     output reg [11:0] addr,
1500     input wire [15:0] data,
1501     input wire start,
1502     input wire last_missing,
1503     output reg [31:0] frame_tdata,
1504     output reg frame_tlast,
1505     input wire frame_tready,
1506     output reg frame_tvalid
1507     );
1508
1509     // Get a signed version of the sample by subtracting half the max
1510     wire signed [15:0] data_signed = {1'b0, data} - (1 << 15);
1511
1512     // SENDING LOGIC
1513     // Once our oversampling is done,
1514     // Start at the frame bram head and send all 4096 buckets of bram.
1515     // Hopefully every time this happens, the FFT core is ready
1516     reg sending = 0;
1517     reg [11:0] send_count = 0;
```

```
1518
1519        always @(posedge clk) begin
1520            frame_tvalid <= 0; // Normally do not send
1521            frame_tlast <= 0; // Normally not the end of a frame
1522            if (!sending) begin
1523                if (start) begin // When a new sample shifts in
1524                    addr <= head; // Start reading at the new head
1525                    send_count <= 0; // Reset send_count
1526                    sending <= 1; // Advance to next state
1527                end
1528            end
1529            else begin
1530                if (last_missing) begin
1531                    // If core thought the frame ended
1532                    sending <= 0; // reset to state 0
1533                end
1534                else begin
1535                    frame_tdata <= {16'b0, data_signed};
1536                    frame_tvalid <= 1; // Signal to fft a sample is ready
1537                    if (frame_tready) begin // If the fft module was ready
1538                        addr <= addr + 1; // Switch to read next sample
1539                        send_count <= send_count + 1; // increment send_count
1540                    end
1541                    if (&send_count) begin
1542                        // We're at last sample
1543                        frame_tlast <= 1; // Tell the core
1544                        if (frame_tready) sending <= 0; // Reset to state 0
1545                    end
1546                end
1547            end
1548        end
1549    end
1550
1551    module debounce #(parameter DELAY=1000000, parameter COUNT=1) (
1552        input wire clk,
1553        input wire reset,
1554        input wire [COUNT-1:0] noisy,
1555        output reg [COUNT-1:0] clean);
1556
1557        genvar i;
1558        generate
1559            for (i = 0; i < COUNT; i = i + 1) begin
1560                reg [19:0] count;
1561                reg new;
1562
1563                always @(posedge clk) begin
1564                    if (reset) begin
1565                        count <= 0;
1566                        new <= noisy[i];
1567                        clean[i] <= noisy[i];
```

```
                end
                else if (noisy[i] != new) begin
                    new <= noisy[i];
                    count <= 0;
                end
                else if (count == DELAY)
                    clean[i] <= new;
                else
                    count <= count+1;
            end
        end
    endgenerate

endmodule

module level_to_pulse (
    input wire clk,
    input wire level,
    output wire pulse);

    reg last_level;
    always @(posedge clk) begin
        last_level <= level;
    end
    assign pulse = level & ~last_level;

endmodule

module display_8hex(
    input wire clk,                    // system clock
    input wire [31:0] data,            // 8 hex numbers, msb first
    output reg [6:0] seg,      // seven segment display output
    output reg [7:0] strobe    // digit strobe
    );

    localparam bits = 13;

    reg [bits:0] counter = 0;  // clear on power up

    wire [6:0] segments[15:0]; // 16 7 bit memorys
    assign segments[0]  = 7'b100_0000;
    assign segments[1]  = 7'b111_1001;
    assign segments[2]  = 7'b010_0100;
    assign segments[3]  = 7'b011_0000;
    assign segments[4]  = 7'b001_1001;
    assign segments[5]  = 7'b001_0010;
    assign segments[6]  = 7'b000_0010;
    assign segments[7]  = 7'b111_1000;
    assign segments[8]  = 7'b000_0000;
    assign segments[9]  = 7'b001_1000;
```

```
1618      assign segments[10] = 7'b000_1000;
1619      assign segments[11] = 7'b000_0011;
1620      assign segments[12] = 7'b010_0111;
1621      assign segments[13] = 7'b010_0001;
1622      assign segments[14] = 7'b000_0110;
1623      assign segments[15] = 7'b000_1110;
1624
1625      always @(posedge clk) begin
1626          counter <= counter + 1;
1627          case (counter[bits:bits-2])
1628              3'b000: begin
1629                  seg <= segments[data[31:28]];
1630                  strobe <= 8'b0111_1111 ;
1631              end
1632              3'b001: begin
1633                  seg <= segments[data[27:24]];
1634                  strobe <= 8'b1011_1111 ;
1635              end
1636              3'b010: begin
1637                  seg <= segments[data[23:20]];
1638                  strobe <= 8'b1101_1111 ;
1639              end
1640              3'b011: begin
1641                  seg <= segments[data[19:16]];
1642                  strobe <= 8'b1110_1111;
1643              end
1644              3'b100: begin
1645                  seg <= segments[data[15:12]];
1646                  strobe <= 8'b1111_0111;
1647              end
1648              3'b101: begin
1649                  seg <= segments[data[11:8]];
1650                  strobe <= 8'b1111_1011;
1651              end
1652              3'b110: begin
1653                  seg <= segments[data[7:4]];
1654                  strobe <= 8'b1111_1101;
1655              end
1656              3'b111: begin
1657                  seg <= segments[data[3:0]];
1658                  strobe <= 8'b1111_1110;
1659              end
1660         endcase
1661      end
1662  endmodule
1663
1664  module xvga(
1665      input wire vclock,
1666      output reg [10:0] hcount,     // pixel number on current line
1667      output reg [9:0] vcount,      // line number
```

```
1668      output reg vsync, hsync, blank);
1669
1670      // horizontal: 1344 pixels total
1671      // display 1024 pixels per line
1672      reg hblank,vblank;
1673      wire hsyncon,hsyncoff,hreset,hblankon;
1674      assign hblankon = (hcount == 1023);
1675      assign hsyncon = (hcount == 1047);
1676      assign hsyncoff = (hcount == 1183);
1677      assign hreset = (hcount == 1343);
1678
1679      // vertical: 806 lines total
1680      // display 768 lines
1681      wire vsyncon,vsyncoff,vreset,vblankon;
1682      assign vblankon = hreset & (vcount == 767);
1683      assign vsyncon = hreset & (vcount == 776);
1684      assign vsyncoff = hreset & (vcount == 782);
1685      assign vreset = hreset & (vcount == 805);
1686
1687      // sync and blanking
1688      wire next_hblank,next_vblank;
1689      assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
1690      assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
1691      always @(posedge vclock) begin
1692          hcount <= hreset ? 0 : hcount + 1;
1693          hblank <= next_hblank;
1694          hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync;  // active low
1695
1696          vcount <= hreset ? (vreset ? 0 : vcount + 1) : vcount;
1697          vblank <= next_vblank;
1698          vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync;  // active low
1699
1700          blank <= next_vblank | (next_hblank & ~hreset);
1701      end
1702  endmodule
1703
1704  module oversample64(
1705      input wire clk,
1706      input wire [11:0] sample,
1707      input wire eoc,
1708      output reg [14:0] oversample,
1709      output reg done
1710      );
1711
1712      reg [5:0] counter = 0;
1713      reg [17:0] accumulator = 0;
1714
1715      always @(posedge clk) begin
1716          done <= 0;
1717          if (eoc) begin
```

```
            // Conversion has ended and we can read a new sample
            if (&counter) begin // If counter is full (64 accumulated)
                // Get final total, divide by 8 with (very limited) rounding.
                oversample <= (accumulator + sample + 3'b100) >> 3;
                done <= 1;
                // Reset accumulator
                accumulator <= 0;
            end
            else begin
                // Else add to accumulator as usual
                accumulator <= accumulator + sample;
                done <= 0;
            end
            counter <= counter + 1;
        end
    end
endmodule
```

B Project Python

```
import random, math, scipy, potrace
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.path as mpath
import matplotlib.patches as mpatches
from tqdm import tqdm
from PIL import Image
from potrace import Curve


# Convert an image to a black and white 2d array (or list for gifs).
def get_bw_file(filename, ratio=128):
    img = Image.open(filename)

    def handle_frame(frame):
        new_data = np.array(frame.getdata(), dtype=np.uint8)
        new_data = np.resize(new_data, (img.size[1], img.size[0]))

        new_data[new_data<ratio] = 0
        new_data[new_data>=ratio] = 1

        return np.pad(
            new_data,
            ((5, 5), (5, 5)),
            'constant',
            constant_values=(1,)
        )

    if img.is_animated:
```

```
        frames = []
        for frame in range(0,img.n_frames):
            img.seek(frame)
            frames.append(handle_frame(img.convert('L')))

        return frames
    else:
        return handle_frame(img.convert('L'))


# Helper function to generate straight line bezier curves.
def straight_line_to_bezier(start, end):
    c1 = tuple(0.666*s + 0.333*e for s,e in zip(start,end))
    c2 = tuple(0.333*s + 0.666*e for s,e in zip(start,end))

    return (start, c1, c2, end)


def greedy_path(lines, rand=False):
    # Greedy pathing: start at an unconnected segment.
    # It has a source and a sink. Choose the closest source or
    # sink, and connect.  Repeat.

    # Distance between two points in 2d
    def dist(a, b):
        return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

    # Calculate the min distance between two paths.
    def calc_distance(a, b):
        return min(
            dist(a[0][0],b[0][0]),
            dist(a[0][0],b[-1][-2]),
            dist(a[-1][-2], b[-1][-2]),
            dist(a[-1][-2],b[0][0])
        )

    # Connect two paths, with freedom to reverse them.
    def connect(a, b):
        d = calc_distance(a,b)

        reverse = lambda x: [
            tuple(reversed(k[:-1]))+(k[-1],)
            for k in reversed(x)
        ]

        if d == dist(a[0][0],b[0][0]):
            return reverse(a) + [straight_line_to_bezier(
                a[0][0],
                b[0][0]
            )+(False,)] + b
```

```
80
81          elif d == dist(a[0][0],b[-1][-2]):
82              return b + [straight_line_to_bezier(
83                  b[-1][-2],
84                  a[0][0]
85              )+(False,)] + a
86
87          elif d == dist(a[-1][-2], b[0][0]):
88              return a + [straight_line_to_bezier(
89                  a[-1][-2],
90                  b[0][0]
91              )+(False,)] + b
92
93          else:
94              return a + [straight_line_to_bezier(
95                  a[-1][-2],
96                  b[-1][-2]
97              )+(False,)] + reverse(b)
98
99      lines = lines[:]
100     for _ in range(len(lines)-1):
101         min_d = 9999999999
102         min_path = None
103
104         focus = random.randrange(len(lines)) if rand else 0
105
106         for other_path in lines:
107             if other_path == lines[focus]:
108                 continue
109
110             d = calc_distance(lines[focus], other_path)
111             if d < min_d:
112                 min_d = d
113                 min_path = other_path
114
115         lines[focus] = connect(lines[focus], min_path)
116         lines.remove(min_path)
117
118     if len(lines) > 0:
119         lines[0] = lines[0] + [straight_line_to_bezier(
120             lines[0][-1][-2],
121             lines[0][0][0]
122         )+(False,)]
123
124         return lines[0]
125
126     return []
127
128
129 # Generate bezier curves and connect them from a BW image.
```

```python
def generate_bezier_curves(image, scale=False, skip=0):
    # Create a bitmap from the array
    bmp = potrace.Bitmap(image)

    # Trace the bitmap to a path
    path = bmp.trace()

    objects = []

    # Generate initial list for each objects
    for i, curve in enumerate(path):
        if i <= skip: continue

        curves = []

        p0 = curve.start_point
        for seg in curve:
            if not seg.is_corner:
                # Is a bezier curve
                curves.append((p0, seg.c1, seg.c2, seg.end_point, True))
            else:
                # Is a corner, convert to two bezier curves (we hacky af)
                curves.append(
                    straight_line_to_bezier(p0, seg.c) + (True, )
                )

                curves.append(
                    straight_line_to_bezier(seg.c, seg.end_point) + (True,)
                )

            p0 = seg.end_point

        objects.append(curves)

    path = greedy_path(objects, rand=True)

    # Get the maximum coordinate, and scale everything
    # such that it equals ~3300.
    b = 3300.0/max(
        max(max(c) for c in curve[:-1])
        for curve in path
    )

    scaled = [
        tuple(
            (b*x,b*y) for x,y in curve[:-1]
        ) + (curve[-1],) for curve in path
    ]

    return scaled
```

```
180
181
182 # Plot a set of bezier curves with matplotlib.
183 def plot_bezier_curves(curves):
184     codes = [
185         mpath.Path.MOVETO,
186         mpath.Path.CURVE4,
187         mpath.Path.CURVE4,
188         mpath.Path.CURVE4
189     ]
190
191     fig = plt.figure()
192     ax = fig.add_subplot(111)
193     for v in curves:
194         if not v[-1]: continue
195
196         patch = mpatches.PathPatch(
197             mpath.Path(list(reversed(v[:-1])),codes),
198             facecolor='none',
199             lw=2, edgecolor=np.random.rand(3,)
200         )
201
202         ax.add_patch(patch)
203
204     ax.set_xlim(0, 4096)
205     ax.set_ylim(0, 4096)
206
207     plt.show()
208
209 # Convert a bezier curve to a .coe row.
210 def bezier_to_binary(curve):
211
212     flattened = (int(round(e)) for t in curve[:-1] for e in t)
213
214     return "".join(
215         "{0:{fill}12b}".format(k, fill='0')
216         for k in flattened
217     ) + ('1' if curve[-1] else '0')
218
219
220 # Output a set of bezier curves to a .coe file.
221 def bezier_to_coe(scenes, filename):
222     instruction_select_bits = int(math.ceil(math.log(
223         max(max(len(frame) for frame in s) for s in scenes)
224     )/math.log(2)))
225
226     frame_select_bits = int(math.ceil(math.log(
227         max(len(s) for s in scenes)
228     )/math.log(2)))
229
```

```
230     scene_select_bits = int(math.ceil(math.log(len(scenes))/math.log(2)))
231
232     total_size = instruction_select_bits+frame_select_bits+scene_select_bits
233
234     print "************************************"
235     print "Scene select bits:", scene_select_bits
236     print "Frame select bits:", frame_select_bits
237     print "Inst select bits:", instruction_select_bits
238
239     buff = "memory_initialization_radix=2;\nmemory_initialization_vector=\n"
240
241     print "Writing COE file now..."
242
243     # Iterate over every entry in our address space.
244     for i in tqdm(range(2**total_size)):
245         s = i >> (total_size - scene_select_bits)
246
247         f = (i >> (instruction_select_bits)) % 2**(frame_select_bits)
248
249         ins = i % 2**(instruction_select_bits)
250
251         scene = scenes[min(s, len(scenes)-1)]
252         frame = scene[f % len(scene)]
253
254         instruction = frame[ins % len(frame)]
255
256         buff += bezier_to_binary(instruction) \
257               + ("," if i != (2**total_size)-1 else ";") + "\n"
258
259     with open(filename, 'w') as f:
260         f.write(buff)
261
262
263 # Slice a bezier curve into two at fraction t.
264 def slice_bezier(points, t):
265     x1, y1 = points[0]
266     x2, y2 = points[1]
267     x3, y3 = points[2]
268     x4, y4 = points[3]
269
270     x12 = (x2-x1)*t+x1
271     y12 = (y2-y1)*t+y1
272
273     x23 = (x3-x2)*t+x2
274     y23 = (y3-y2)*t+y2
275
276     x34 = (x4-x3)*t+x3
277     y34 = (y4-y3)*t+y3
278
279     x123 = (x23-x12)*t+x12
```

```
    y123 = (y23-y12)*t+y12

    x234 = (x34-x23)*t+x23
    y234 = (y34-y23)*t+y23

    x1234 = (x234-x123)*t+x123
    y1234 = (y234-y123)*t+y123

    return (
        ((x1,y1),(x12,y12),(x123,y123),(x1234,y1234), points[-1]),
        ((x1234,y1234),(x234,y234),(x34,y34),(x4,y4), points[-1])
    )


# Numerically compute the length of a bezier curve.  Turns out doing this
# "prettier ways" is hard af.
def bezier_length(curve):
    # Split curve into 1000 points, and find their sum distance

    return sum(
        sum((first[k] - second[k])**2 for k in (0,1))
        for first, second in (lambda x: zip(x,x[1:]))([
            tuple(
                (1-t)**3 * c[0] + 3*t*(1-t)**2 * c[1] +
                3 * t**2 * (1-t) * c[2] + t**3 * c[3]
                for c in zip(*curve[:-1])
            ) for t in np.linspace(0,1,1000)
        ])
    )


# Expand a set of bezier curves such that they are a power of two
# by splitting the largest curves in half.
def expand_bezier(curves):
    target = int(2**math.ceil(math.log(len(curves))/math.log(2)))

    c = {curve: bezier_length(curve) for curve in curves}
    for _ in range(target-len(curves)):
        # Pick the biggest curve, and split it down the middle.

        max_curve, max_len, max_index = None, -1, -1

        for i, curve in enumerate(curves):
            l = c[curve]


            if l > max_len:
                max_len = l
                max_curve = curve
                max_index = i
```

```
330
331         first, second = slice_bezier(max_curve, 0.5)
332         curves[max_index] = first
333         curves.insert(max_index+1, second)
334
335         c[first] = c[second] = 1/2
336
337     return curves
338
339
340 # Apply the entire processing pipeline to a single gif.
341 def process_gif(filename,
342                 down_sample=1,
343                 skip=0,
344                 get_first=False,
345                 pad=0,
346                 ratio=128):
347
348     print "Getting␣images␣from", filename
349     imgs = get_bw_file(filename, ratio=ratio)
350
351
352     imgs = [imgs[0]] if get_first else [
353         imgs[i] for i in range(0,len(imgs),down_sample)
354     ]
355
356     imgs = imgs[pad:0-pad-1] + ([imgs[-1]] if pad == 0 else [])
357
358     print "Processing␣initial␣bezier␣curves..."
359     frames = [generate_bezier_curves(x,scale=True,skip=skip) for x in imgs]
360
361     print "Expanding␣bezier␣curves..."
362     frames = [expand_bezier(x) for x in frames]
363
364     instruction_select_bits = int(math.ceil(math.log(
365         max(len(f) for f in frames)
366     )/math.log(2)))
367
368     frame_select_bits = int(math.ceil(math.log(len(frames))/math.log(2)))
369
370     return frames
371
372
373 # The final COE used during our checkoff + video.
374 saved = [
375     process_gif('macaroni.gif'),
376     process_gif('pear.gif',skip=1),
377     process_gif('square.gif',down_sample=3),
378     process_gif('dots.gif',skip=1,down_sample=3)
379 ]
```

```
380
381 bezier_to_coe(saved, "rom.coe")
```