‘

3D Music Visualizer

Miguel Gomez, Nancy Vargas

6.111 Fall 2020

Table Of Contents

Overview

When listening to music, it's often really nice to have a visual experience that accompanies it as well. To achieve this, music visualizers analyze music as it is playing, and extract certain features or patterns to then translate into some visual effect and display to the user. This is not a new concept, and has existed since computing hardware became popular for the general public, beginning with the Atari Video Music in 1976. Over the years, many more music visualizers have been released, such as those included with Windows Media Player in Windows XP, or Winamp, which was pretty widely used.

This project aims to take this concept and apply it using an FPGA, displaying a 3D rendering of a sphere that transforms itself in sync with the music. Most of the time, the sphere rotates along 3 axes with different rates with a set color. With each detected beat in the music, the sphere expands and contracts with a reverb effect, changes color. and changes the direction of rotation. The sphere itself is a collection of 300 points which move around in unison.

Figure 1: Example frame from final result

Audio Pipeline

Overview

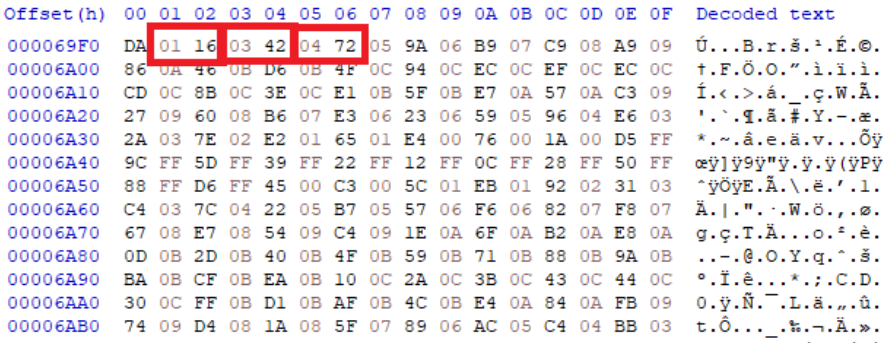
The audio pipeline consists of all the modules that collectively take the music, analyze it, and produce the beat signal that is then used by the graphics pipeline to create the visual representation. The SD card reader first takes in music data from the SD card and stores it into a block of BRAM. The sampler then reads data from the BRAM at the appropriate rate, and sends each sample of data to the PWM module and the FFT pipeline. The PWM module takes the sample and sends a generated signal to the PWM output, which is an everyday ⅛ inch headphone jack connected to a speaker. The FFT pipeline processes the music and uses the result to analyze the onset of a beat. It finally outputs this signal, which is used by the graphics pipeline. In addition, it visually outputs part of the FFT results through the LEDs on the FPGA. The block diagram for this section can be found in Appendix A.

SD Card Reader

sd_top.sv

The SD card reader takes in a read signal from the sampler, and proceeds to read SD card data when it is asserted, storing it into BRAM. This module is run at 25Mhz, which is different from the rest of the design, so the read signal must be synchronized before being used by the module. To actually read the data, we use the given sd_controller module, which handles all the actual control signals. The greatest amount of work here went into ensuring that the 16-bit samples are read and stored correctly for the sampler to use (reasons for using 16 bit samples are explained in the FFT section). To do this, we implemented a counter that counts up to 2048, which is the number of bytes we need to read per FFT window. We begin the counter at 0, and when the read signal is asserted by the sampler, we send the read signal to the controller, and begin receiving the bytes. For every two bytes received, these are grouped into one 16-bit

sample and stored into the BRAM at address count/2. As each byte is received, the counter increases and once it hits 511, it sends another read signal immediately, so we read 1024 bytes (or 512 samples) total. Once the count is 1023, it waits for another read signal from the sampler to repeat the whole process, just writing to a different location in memory. This is so that the sampler can read previous samples while the reader writes the next samples, without interfering with each other. (A lot of the structure here could be simplified or seems unnecessary because originally it was planned for 8 bit samples, and was modified very quickly to work with 16 bits.)



Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
000069F0	DA	01	16	03	42	04	72	05	9A	06	B9	07	C9	08	A9	09	Ú...B.r.š.¹.É.©.
00006A00	86	0A	46	0B	D6	0B	4F	0C	94	0C	EC	0C	EF	0C	EC	0C	†.F.Ö.O.”.ì.ï.ì.
00006A10	CD	0C	8B	0C	3E	0C	E1	0B	5F	0B	E7	0A	57	0A	C3	09	Í.‹.>.á._.ç.W.Ã.
00006A20	27	09	60	08	B6	07	E3	06	23	06	59	05	96	04	E6	03	'.`.¶.ã.#.Y.–.æ.
00006A30	2A	03	7E	02	E2	01	65	01	E4	00	76	00	1A	00	D5	FF	*.~.â.e.ä.v...Õÿ
00006A40	9C	FF	5D	FF	39	FF	22	FF	12	FF	0C	FF	28	FF	50	FF	œÿ]ÿ9ÿ"ÿ.ÿ.ÿ(ÿPÿ
00006A50	88	FF	D6	FF	45	00	C3	00	5C	01	EB	01	92	02	31	03	ˆÿÖÿE.Ã.\.ë.’.1.
00006A60	C4	03	7C	04	22	05	B7	05	57	06	F6	06	82	07	F8	07	Ä.|."..·.W.ö.‚.ø.
00006A70	67	08	E7	08	54	09	C4	09	1E	0A	6F	0A	B2	0A	E8	0A	g.ç.T.Ä...o.².è.
00006A80	0D	0B	2D	0B	40	0B	4F	0B	59	0B	71	0B	88	0B	9A	0B	..-.@.O.Y.q.ˆ.š.
00006A90	BA	0B	CF	0B	EA	0B	10	0C	2A	0C	3B	0C	43	0C	44	0C	º.Ï.ê...*.;.C.D.
00006AA0	30	0C	FF	0B	D1	0B	AF	0B	4C	0B	E4	0A	84	0A	FB	09	0.ÿ.Ñ.¯.L.ä.„.û.
00006AB0	74	09	D4	08	1A	08	5F	07	89	06	AC	05	C4	04	BB	03	t.Ô..._.‰.¬.Ä.».

Figure 2: Samples in WAV file

In addition, we also took input from the up, left, center, and right directional buttons (down was reserved for reset) to allow the selection of different songs. Those songs were Dirt Off Your Shoulder (Jay-Z), Mercy (Kanye West), Animals (Martin Garrix), and Pursuit of Happiness (Kid Cudi, Steve Aoki Remix). The reason for selecting these songs is due to the frequency range that the beats are present in, as well as the fact that these songs are absolute bangers that I will gladly black out to at frat parties.

Sampler

sampler.sv

The sampler is a very simple module, since at this point the hard work of getting the data into nice 16 bit samples has been accomplished. This module simply consists of a sample counter which counts up to 2268 (1 divided by the sample rate of 44.1kHz), as well as an address counter which counts up to 1023. Every time the sample counter reaches the max, it reads a new sample from the BRAM, outputs it to the top module

(where it is then used by the PWM module and the FFT Pipeline), and increments the address counter by one so it reads a new sample the next time. When the address counter reaches 511 or 1023, it sends the read signal to the SD card reader to signal that it should start reading more samples and writing to the section it just finished reading.

FFT Pipeline, PWM, and LED Manager

Not their own files, implemented in audio_top.sv

The sample from the sampler has to be modified first by flipping the first bit, since it's originally signed. After this, it goes to two places: the PWM module and the FFT pipeline.

The PWM module only takes the first 8 bits, and outputs a PWM signal where the percentage of time on corresponds to the value of the sample (this is the same module from 5a, not modified), which goes directly to the speaker.

The full 16-bit signal is sent to the FFT pipeline, which analyzes a 1024 sample window of sound and splits the sound in each window into bands of separate frequencies. 16 bits per sample were needed because with 8 bits, much of the signal information is lost, and the final FFT results are very rough and difficult to get any beat data from. The FFT pipeline mostly consists of the example modules provided, which include the FFT, square and sum, FIFO, and square root modules. After receiving the output from the square root module, we are only interested in band 0, since this is the most common band where beat information can be detected easily. There is of course music where the beat doesn't lie in the bass, however for this project we aimed to mainly focus on the bass frequencies, and used such music to test.

With band 0 extracted, the LED manager simply takes the band audio level and lights up the LEDs from 0 to *n*, where *n* is the floor of the audio level divided by 1000. This turns the LEDs into a live audio meter for the bass frequencies, as shown in the figure below.

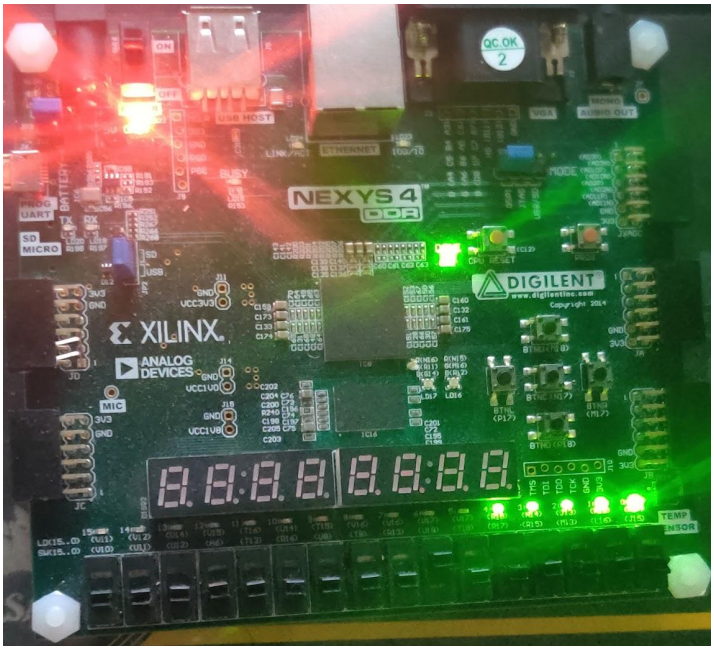


Figure 3: LED Audio Level

Beat Detector

beat_det.sv

The beat detector takes the band 0 level from the FFT pipeline and a threshold level, and outputs a single-cycle beat signal to be used by the graphics pipeline. The threshold level is set by the switches on the board (which conveniently line up with the LEDs). The threshold is *n**1000, where *n* is the index of the highest switch that is flipped up. The way it predicts a beat is relatively straightforward. If the audio level rises, passing this threshold, it sends the beat signal out. (It's easily visualized using the LEDs, since the beat signal is sent out whenever the LED level rises above the highest flipped switch). There's also a slight delay of 5 FFT windows before the beat can be reactivated, to try to minimize the impact of the noise. Although this beat detection method isn't really that sophisticated, it works well enough for a lot of songs, as shown in the demo video.

Graphics Pipeline

Overview

The graphics pipeline takes in the beat signal generated by the audio pipeline, and generates a video signal to represent the music. First, the parameter generator takes in the beat signal, and generates 3 parameters which will be used to transform the sphere points: sine, cosine, and scale. These are passed into the vertex transformation module, which reads a precomputed set of points, and transforms them according to these parameters. These points are then projected onto a 2d plane and outputted into the signal control module, which then writes the correct color data into one of two BRAM blocks. The color is determined by an FSM depending only on the current color, which results in a sequence that is repeated. Finally, the renderer reads the correct pixel from one of the BRAM blocks, and outputs it to the screen. The block diagram for the graphics pipeline can be found in Appendix A.

Parameter Generator

angle_gen.sv

The parameter generator takes in the beat signal and converted vsync signal (represented as a single pulse whenever there is a new frame), and outputs a sine, cosine, and scale by which the points have to be transformed by the transformation module. In this module, we have a state consisting of 3 angles by which the points are rotated by (pitch, yaw, roll). Every time the converted vsync signal is asserted, the angles get incremented by a constant amount determined by a rate of change. This means that every frame, the points rotate by a given amount, regardless of the presence of a beat. When a beat is asserted, we change the rate by negating some of the directions, so that the new rate causes the points to rotate in a new direction. Every frame, we run the current angles through CORDIC modules to calculate the sine and cosine for all of these, which are fed into the transformation module.

Additionally, we output a scale factor to resize the sphere in sync with the beat. When the beat is asserted, we begin outputting a precomputed sequence of scale factors which are supposed to roughly follow a sine wave that is gradually damped, as shown below.

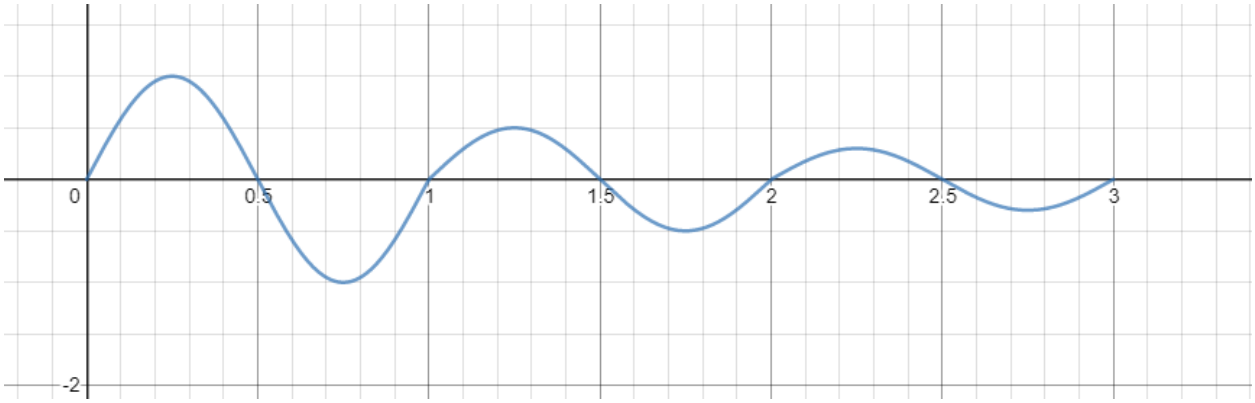


Figure 4: Pseudo-damped sine wave

Vertex Transformation

transformation.sv

Overview

The vertex transformation module consists of 4 main submodules: rotation, scale, translation, and projection. Essentially, this module takes the precomputed points from the BRAM and transforms them multiple times via matrix multiplications, before outputting the final points in terms of pixel coordinates on a display.

Point calculation

points.py

The point calculations were done using a python script, which essentially randomly picks certain parameters and generates points with coordinates that lie on the surface of a sphere with radius 1024. The script takes these coordinates, and then does some formatting to obtain the binary representations and write it to a COE file with

proper headings. This was really convenient towards the end, since to move from 30 to 300 points only required a single parameter change.

Rotation, Scale, Translation

rotation.sv, scale.sv, translation.sv

These submodules take in the sine, cosine, and scale calculations from the previous module, and transform the points directly read from the BRAM one at a time, applying the matrix multiplications shown below for rotation and scaling (for translation, we simply add the translation amount to the vertices).

$$R = R_z(\alpha)\,R_y(\beta)\,R_x(\gamma) = \underbrace{\begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}}_{\text{yaw}} \underbrace{\begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix}}_{\text{pitch}} \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & \sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix}}_{\text{roll}}$$

$$S_v = \begin{bmatrix} v_x & 0 & 0 \\ 0 & v_y & 0 \\ 0 & 0 & v_z \end{bmatrix}.$$

Projection

projection.sv

The projection module takes in the coordinates, and applies one last matrix transformation to project the 3D coordinates onto a 2D plane with depth data (although in the end the depth data was not used, as overlapping points simply overwrite each other, since they are the same color in the final result). Specifically, we used perspective projection, which takes into account the viewer angle, as opposed to orthogonal projection, which takes away perspective (i.e., objects don't get smaller as they get farther away). This projection allows the sphere to look real, since closer points seem to

move less and further points move faster. However, this required division within the matrix multiplication, so we had to use some division modules.

$$M_{proj} = \begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & -\frac{zFar + zNear}{zFar - zNear} & -\frac{2 \cdot zFar \cdot zNear}{zFar - zNear} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

We used the aspect ratio corresponding to a 4:3 screen, and the camera FOV set to 90 degrees.

Signal Control

In graphics_top.sv

The signal control module handles a variety of signals, converting from different clock domains, as well as sending the control signals to write to the correct addresses in BRAM for the renderer to read. First, this module calculates its own vsync, hcount, and count signals which are slightly different from the default xvga ones, to take advantage of the time during the front, back porch, and sync sections of the signal and allow calculations to happen more efficiently (for example, in the xvga module, hcount and vcount aren't updated until just before the new line is being read, while in the new signals, these are updated as soon as the last pixel on the previous line has been read).

In addition, it takes care of writing the pixel data to one of two BRAMs. These BRAMs each use one 18K block and one 36K block, and are enough to store pixel data for the entire screen by staggering the pixel data. Each one stores 4 horizontal lines of pixel data, and whenever hcount is increased past the bounds of the current BRAM, a signal is sent to recalculate all the points and write pixel data for the next 4 lines into the BRAM that currently isn't being read by the renderer. Within the time it takes to

cover 4 lines of pixel data, the graphics module has time to recalculate all the points, and only store those into BRAM that lie within the next 4-line section of the screen. To write to the correct address, it takes the output coordinates and writes to BRAM address y[1:0]*640 + x, checking first that the entire y-coordinate lies within the next section.

Renderer

In graphics_top.sv

The renderer is pretty straightforward as well. All it does is take hcount and vcount, look at the 2nd bit of vcount (indexed from 0) to decide from which BRAM to read, and output that to the screen. After reading each pixel, it writes a 0 to each location, so that if a pixel location isn't written by the previous module, it defaults to a black screen.

Challenges and Conclusion

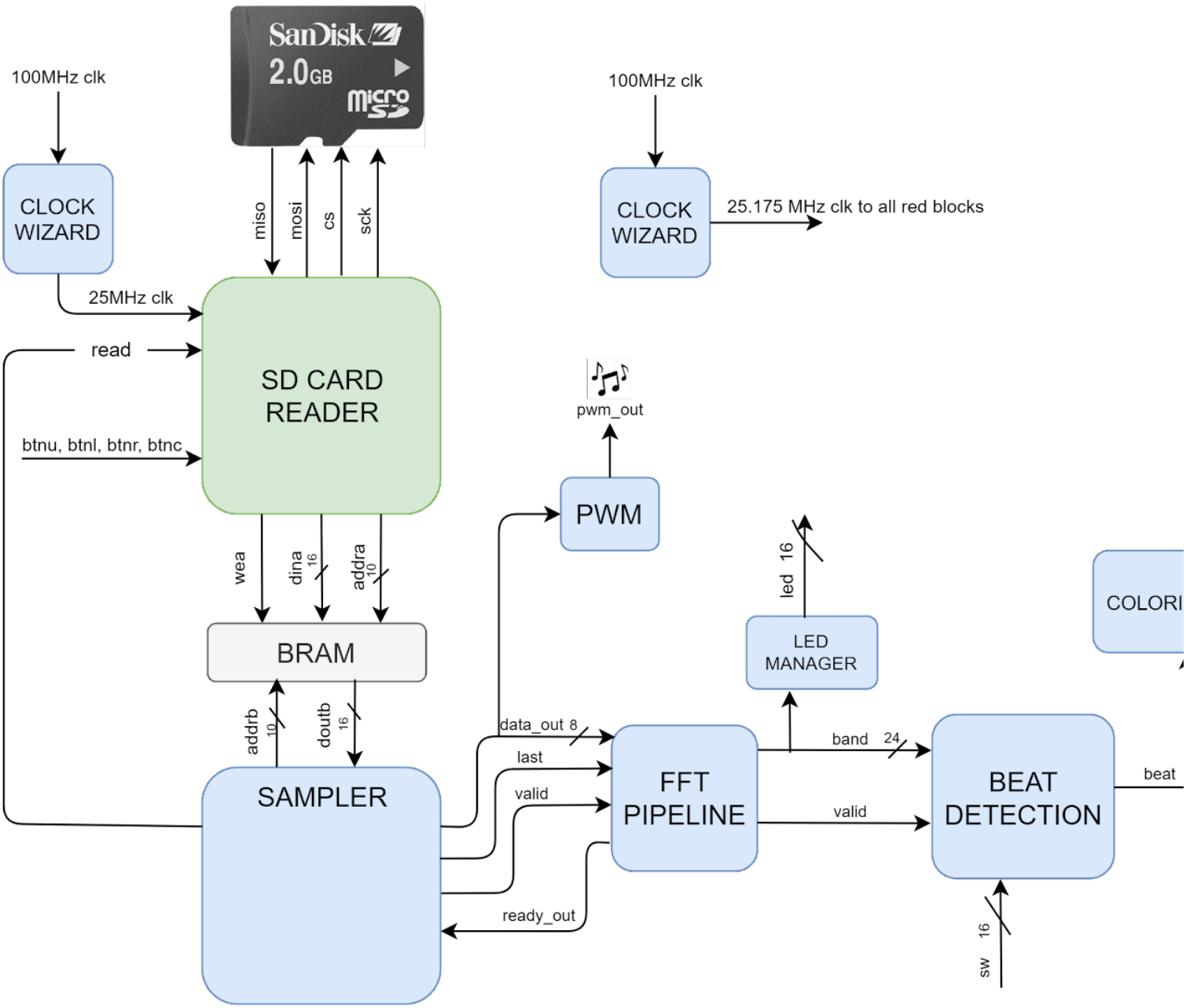
There were many aspects that were pretty difficult, especially along the graphics pipeline side. We essentially had to speedrun a computer graphics course to understand how to transform the points and project them properly, while managing the very small amount of memory resources on the board, and handling clock domain synchronization for various signals.

The audio pipeline was also challenging at times, such as modifying it to take 16 bit samples rather than 8 bit. This required a lot of delicate synchronization between the reader and sampler modules to read and sample at the appropriate rates while preventing them from stepping on each others' toes in reading/writing from the BRAM. Also, the FFT output was not the cleanest and it seemed to sometimes normalize the band data during quiet sections, which was unwanted and posed a bit of a problem (I'm not sure if this was supposed to happen, or if it was something with the implementation, but we worked around it).

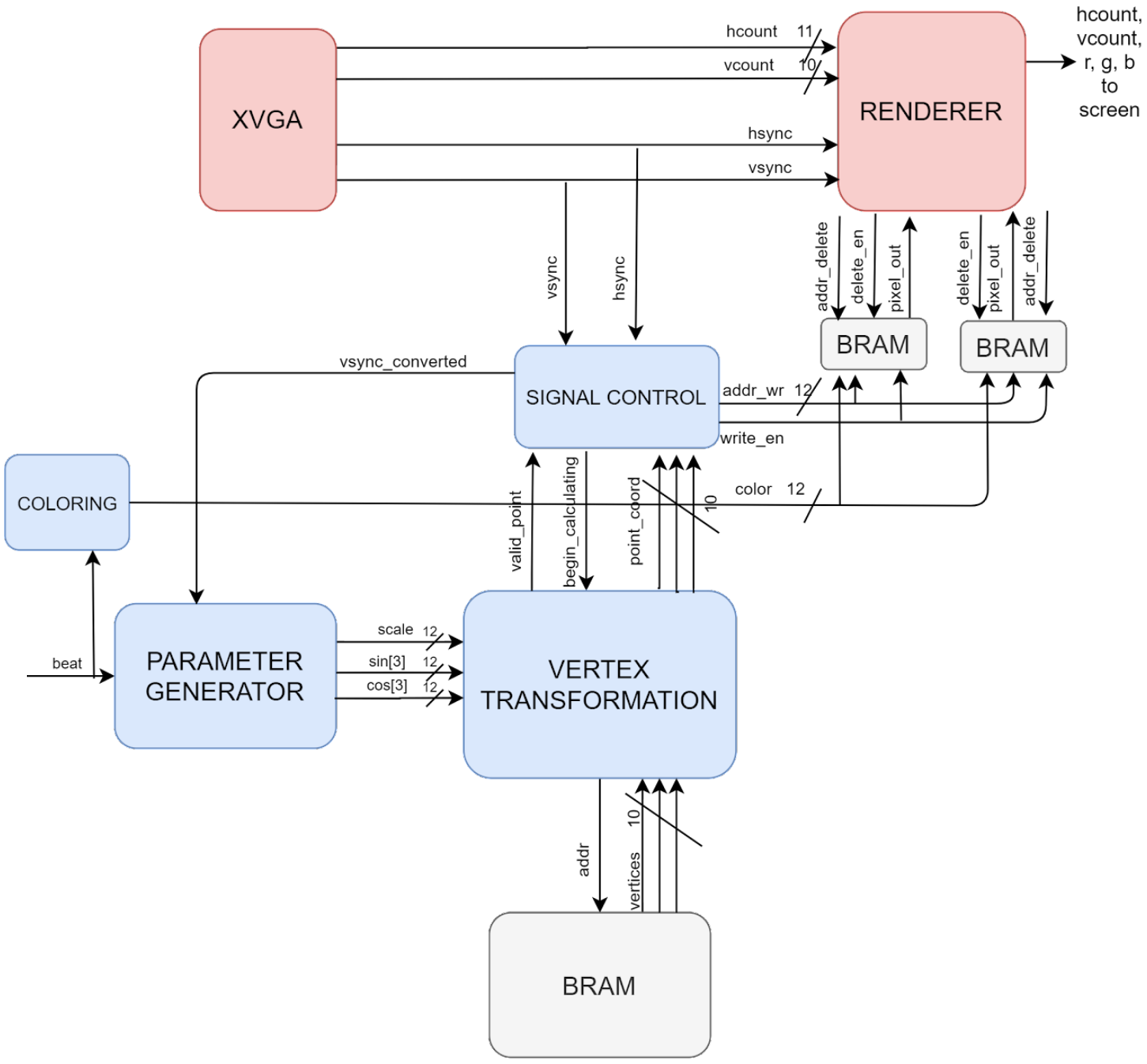
Overall, the project turned out really well, even with the unsophisticated beat detection method, and is able to visualize a wide variety of music in a visually pleasing way.

Appendix A - Block Diagrams

Audio Pipeline



Graphics Pipeline



Appendix B - Source Code

Source code can be found at: https://github.com/mgomez12/3DVisualizer