Beat Saber Final Project Report

Shreya Karpoor
Department of EECS
Massachusetts Institute of Technology
skarpoor@mit.edu

Darren Lim
Department of EECS
Massachusetts Institute of Technology
darrenl@mit.edu

Monica Liu
Department of EECS
Massachusetts Institute of Technology
mqliu@mit.edu

***Abstract*—Below, we outline our preliminary report for work done thus far in the final project for the Huah Huah Huah group. We can divide up the work into the following three sections, which is roughly divided among the team members: Image Processing, Game State/Rendering, and Music Processing. The Image Processing modules aim to use the camera to determine hand and head positions in a 3D space. The Game State modules aim to keep track of the state of the game, and the Rendering modules aim to use the VGA to render the entire game state in either 2D or 3D using raycasting. Finally, the Music Processing modules aim to play a music wav file from an SD card and synch music output with game play. The Music processing modules will also initialize the block locations based on music characteristics like changes in amplitude and frquency.**

I. Introduction

Our goal is to create an augmented reality Beat Saber game implementation using the FPGA. In our project, we aim to create a seamless user experience that uses the parallelizability of the FPGA to render a 3D game in real-time. Our main goals are to learn about how to project 3D shapes onto a 2D plane using raycasting, play and communicate music synchronization on the FPGA, track hand and head positions and velocities, and keep track of a big game state to manage all of these different components during the entire game.b

II. Game Logic (Darren)

The game logic follows a simple state machine, which stores the current player's score, most recently sliced blocks, and the current time. The game runs on a current time counter, which moves once every 10 milliseconds. This current time counter was chosen in order to give sufficient time for the rendering modules, as described below in more detail, to update the framebuffer. The game logic also ensures that blocks are loaded in order, the saber's velocity is accounted for, block positions are calculated, and blocks are sliced correctly. Below, we will go over the modules in more detail. See the `game_logic_and_renderer.sv` diagram for a overview of the game state and how it interfaces with the renderer.

A. Game State

The game state outputs the current time and the player's current score, the latter of which is outputted on the LED array on the FPGA. The current time drives all other modules downstream.

B. Block Loader

The block loader reads block data preloaded in a .mem file, which stores the following block information for every block: X position, Y position, time to be hit (in the units defined above), color (red or blue), and direction (up, left, right, down). We use a read-only BRAM to read the .mem file, and load the blocks in order of time. Since we know that blocks will only appear in increasing time order, we can stream data from the BRAM as a cache. We output 12 block data at a time, where the first index is the block that is closest to being hit, and the last index is the farthest block. In our demo, we only need to render at most 5-6 blocks at a time, but theoretically up to 12 can be done using this method. This block information is used downstream, especially in the block selector module, to figure out what block is being rendered for a particular XY position on the monitor.

C. Saber History

The saber history stores the previous saber position offset by a constant time. This allows us to calculate velocity, which is the difference in saber positions between time periods. This is used downstream in the state processor to determine if blocks have been sliced in the correct direction.

D. Block Positions

The block positions module will take in all the block data from the block loader and augment this data with the Z position, calculated from the time that the block will hit the player, and whether or not the block is currently visible. Since the block loader loads the 12 next blocks from the BRAM, not all blocks will be visible until it reaches a certain time threshold.

E. State Processor

The state processor will incorporate the information from the saber history and output whether a block has been sliced. Since we can only slice the block that is closest to the player from the block loader, we only check this one block for whether this condition holds.

III. Music Synchronization (Shreya)

A. Block Generation

Currently, audio preprocessing for block generation is done in python. Audio amplitude is analyzed to determine block positions that match with the song's inherent beat. This allows

custom songs to be loaded onto the SD card and used for game play. Currently, the blocks that are generated need to be manually loaded as BRAM onto FPGA 1 since audio preprocessing occurs on our local PCs, but future work can implement this processing using Verilog and use our already existing Serial module to transmit block hex data from FPGA 2 to FPGA 1. The `block_gen` logic, however, is the same and is described below.

For pre-loaded songs, we have hard coded block positions. For each block, we specify (x, y, z, t) for block starting position. This information is stored as a mem file and sent via serial to our second FPGA that does game logic and rendering prior to game start.

For custom songs, we create a mem file with block starting position based on song characteristics. Below is an explanation of our analysis in Python.

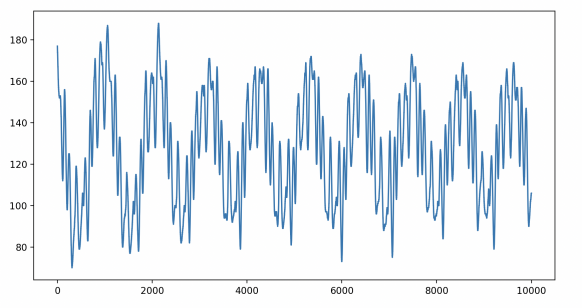

Fig. 1. Sample from raw wave file

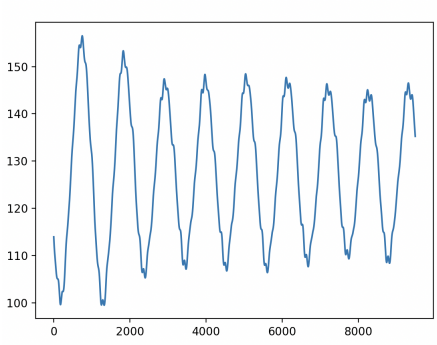

Fig. 2. Smoothing of raw wave file

Figures 2-4 show the steps to our custom song analysis. Wave file data is first averaged with a 1D convolution to achieve the smoothing effect in Figure 3. Figure 4 is then a derivative of the amplitudes seen in Figure 3. This is computed with another 1D convolution with a kernel of $[-1, 0, ..., 1]$ of length N=5000 in this example. N was set as approximately $\frac{8}{\text{sample rate}} = \frac{44100}{8}$. This second 1D convolution was done to detect steep rises in amplitude shown in Figure 4. Blocks will be generated with a random (x, y, z) at every rising edge of the wave function shown in Figure 4. Block generation is done prior to game play, so any latency here will not affect music and game rendering synchronization.

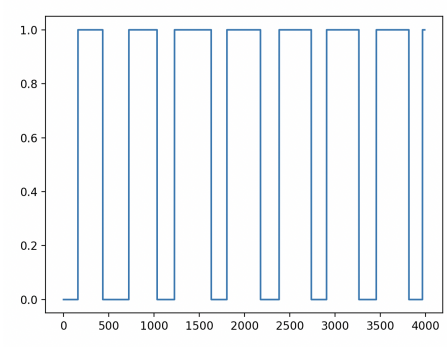

Fig. 3. Step function corresponding to amplitude peaks

B. Audio Pipeline

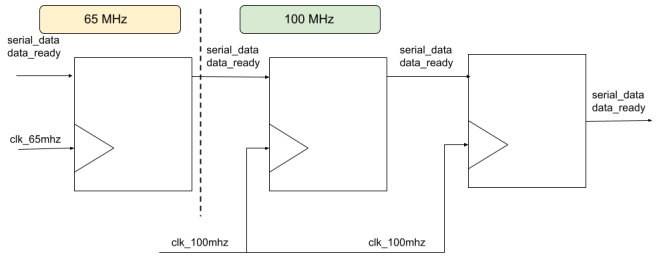

Fig. 4. Flip flop synchronization from 65 MHz to 100 MHz clock domain

The Figure 9 shows the high level timing of the audio pipeline. Three different clocks were used because of constraints to the `sd_controller`, `audio_pwm`, and VGA modules. The ordering of the pipeline was optimized to try and have most domains from a slower to a faster clock frequency. This is so that a 2 flip flop synchronizer shown in Figure 4 can be used. This allows the registers to be able to catch the rising edge of the slower clock since the register operates in a much faster clock frequency. The one cross from 100 MHz to 25 MHz is for the start byte from FPGA 1 that signifies game start. To ensure that the sd_read_state_control module catches the game start byte, we transmit the start byte multiple times (100 times) and game state begins. This doesn't affect game play because once the byte is received, the state moves to the `sd_read` state and is no more scanning the receiver for the `start_byte`. Overall timing issues/resolutions will be described in the subsections within the music generation section.

C. SD Card Reading

Songs are read from an SD card on board the Nexys 4 and played to a speaker output. Wav files were directly written to the SD card in hex format and separated by 1000 bytes from each other. The SD card module, given in the 6.205 Documentation page, reads music data in 512 byte chunks, and is currently set to start reading music data from byte 512 of the desired wav file. The first 512 bytes of data are part of

the wav file header specifying wav file length, sampling rate, number of channels, etc... Reading music data is done at 25 MHz as specified in the module documentation.

Communication between the external SD card and the Nexys board is done using SPI. SPI is a common interface used for communication between a microcontroller and its peripherals. The 4-wire SPI uses a CS (chip select) register set to the SD peripheral, a shared clock CLK generated by the Nexys, an input MISO (main out, subnode in), and an output MOSI (main in, subnode out). Here, we use the MOSI line to transmit data from the subnode SD card to the Nexys.

The `sd_controller` module utilizes the SPI interface described above to read the data written in the SD card to the main board. Data is made available 1 byte at a time, and is read continuously in chunks of 512 bytes. With a 25 MHz clock, 512 bytes of data is 1/25000000*512 seconds of music, which is pretty short. Data is made available on the rising edge of the `data_available` signal. Once a 512 byte read operation is done, ready is asserted and the module is ready to read another 512 bytes starting at the specified SD address. This functionality was exploited to start and stop SD card reading to ensure correct data is available for the speaker output. Reading from the SD card is completed once the address reaches the last byte address of the wav file. The length of the wav file was measured prior to processing, and is also made available by storing the 4 bytes that occur after the 32'h64_61_74_61 in the wav file header at the beginning of the file.

Using this module was not too difficult; however, interfacing this module with the complete audio pipeline took more time than expected due to clock domain crossing and other timing considerations that will be explained in the next sections.

D. Speaker Output

Data read from the SD card is written to an `audio_pwm` module, also given in the 6.205 Documentation page, at a 44.1KHz frequency. So, SD card reading is done about 1e03 times faster than audio signal generation. Therefore, simply reading from the SD card and outputting the data to speaker output leads to a way faster sampling rate (found from experience).

First, I implemented a large amount of BRAM to try and solve this problem, but later discovered the First In First Out (FIFO) IP that does the same thing and is much easier to integrate and debug. The FIFO acts as a buffer that can be written to and read from. In this case, data is written and read from in 1 byte sections. Data read from the SD card is written to the FIFO and when the data from the FIFO is read from the FIFO, it is written to the speaker output modules.

The FIFO used has a common 25 MHz clock and a width of 16384. Reading from the FIFO is done every 568 clock cycles while writing occurs every 20 clock cycles, as specified by the `sd_controller` module. 568 was chosen to divide the 25 MHz clock signal to a 44KHz signal to achieve the desired sampling rate of $\frac{25e06}{44e03} \approx 568$

This means the FIFO is likely to get full because of the large between the read and write rates. The issue with this, which I discovered while debugging, is that FIFO writing is halted when it gets full but SD card reading is not halted unless the rd line in the `sd_controller` module is manually driven low. When this happens, audio data is skipped since data that is being read is not being stored anywhere.

To solve this, when the FIFO gets close to being full (with less than 1536 empty entries available for writing), SD card reading is halted by disabling the rd line while the FIFO reading continues to allow the FIFO to become more empty and allow writing to the FIFO.

This was probably the most challenging part of the audio integration. I initially tried just the approach described above, but the audio sounded very fuzzy. There were a few reasons for the background noise. One, is that I was using the `audio_pwm` module with a 25MHz clock instead of the 100MHz specified by `audio_pwm` documentation.

To get rid of some fuzziness, I implemented clock domain crossing from 25 MHz to 100MHz, and integrated `audio_pwm` for speaker output with 100MHz. This was done using a 2 flip flop synchronizer, shown in the Figure 4. Initially, I was seeing some glitches, and added a second register to decrease the risk of metastability.

The final sound quality, as seen in the final report video, is much cleaner than initial iterations. There are a few reasons for this. I found, using the ILA, that I am sampling the music data at the correct 44KHz; however, the number of entries being written to the FIFO needed to be increased. As described earlier, SD card reading is much faster than writing to the `audio_pwm` controller. Stopping reading from the SD card reader was initially being done when the FIFO had more than 1024 entries and would restart when the FIFO had only 512 entries, meaning 512 bytes, or 1 SD card read. Instead, I increased the upper bound to stopping SD card reading when the FIFO was more than 1536 entries, so the FIFO now has room for 2 SD card reads. This increase in the usable section of the FIFO greatly improved audio quality. Finally, I preprocessed my audio data to include both a low pass filter below the 22KHz Nyquist frequency (which didn't make a huge difference considering that is on the upper range of the human hearing range) and more importantly, reduced the gain of the whole audio file. This ensured that there was absolutely no clipping in any part of the audio.

E. Serial UART

Serial UART is a method of communication that was implemented in this project for communication between our two FPGAs. It uses the Rx and Tx lines that allow for two way communication. This allows both FPGAs to both transmit and receive data so long as both transmitter and receiver share a clock and have the same baud rate.

BaudGen The `baudGen` module works by raising dividing a given clock to obtain an 115200 baud rate. We divide a 65 MHz clock to get 115200 baud rate, by using an accumulator of size 2^{14} and incrementing by 29 every clock cycle:

$$\frac{2^{14}}{19} \approx \frac{6500000}{115200}$$

Transmitter The transmitter serializes data. This means it takes one byte of data and adds a start and stop bit. The Tx line stays high in idle state, drops low for one baud cycle to indicate start of transmission, sends each bit of data on each baud cycle, and a stop bit of value 1 for another baud cycle.

Receiver The receiver deserializes the data. It takes the input from the transmission output and watches for a start low bit. Then it stores the next 8 bits in a buffer stored over the next 8 baud bits, and checks for a correct stop bit. If the stop bit is not received, it clears the buffer to ensure false data is not received and used.

FPGA to FPGA Implementation (Monica) To implement serial communication between two FPGAs, one FPGA implements the transmitter module and the other implements the receiver module. In terms of wiring, the transmitting FPGA must tie TxD to a port as an output (for example set output logic jc[0]) in its top level. The receiving FPGA must tie RxD to a port as an input (for example, input wire jc[0]) in its top level. Finally, the two FPGAs achieve a common ground and signal with a physical wire connecting their ground ports together and a physical wire connecting the TxD port to the RxD port.

The two FPGAs are connected now in hardware and use the same baud rate, however, their clocks are not synchronized and thus may differ in timing by an unknown phase shift. As a result, the receiver would sample close to the edges of the transmitter signal and interpret the data incorrectly. To address this miscommunication, we modify the receiver. When the receiver is idle and detects a falling edge indicating the start bit, it begins counting up to half a baud cycle. Once a half cycle has passed (282 cycles on a 65MHz clock is half of 1/115200), then the receiver checks again if the start bit is still low (to verify that the falling edge was not a glitch). If the line is still low, then the receiver resets its baud so that it's offset approximately half a cycle from the transmitter's baud. With the offset, even if the clocks skew from each other by a small margin, the receiver will still receive the correct bit.

F. FPGA Communication for Audio (Shreya)

To start playing the song, FPGA 1 sends a byte of all ones to FPGA 2. When the receiver on FPGA2 receives this data, it starts reading audio data from the SD card and feeding it to the `audio_pwm` module.

Another feature of the music jukebox is that it plays a 740 Hz PWM wave for about 0.1 sec when a block is correctly sliced. Hit detection is done in the game logic modules. When a hit is recognized, FPGA 1 sends a byte of data via serial to FPGA 2. When the receiver on FPGA 2 receives and deserializes the data, the `audio_pwm` module receives the 740 Hz PWM wave audio data instead of the SD card for 0.1 sec.

The delay between the first button press to the first note played on the speaker and the delay between a hit on the screen and the hit note played on the speaker is 568 clock cycles in the 25 MHz domain and 10 cycles with a 115200 baud rate. This difference is approximately 3.14e-5 seconds and is negligible to the human ear. We tested this, and in our final video, it can be seen that many of the blocks appear at a down beat in the song (not all down beats since we reduced frequency of blocks to make game play slightly easier) and the hit sound is heard almost instantaneously to the human ear after a hit is rendered on VGA.

G. Insights (Shreya)

I learned many many things, some of which I described earlier. The key takeaways were to test every step of the way. First, I would test via testbenching, such as when testing if serializing was working, and then with the ILA if I was interfacing with hardware that I could not scope with the oscilloscope or test through testbenching. Additionally, working with domain crossing and FIFO's was a learning experience. I definitely understood the importance of timing, metastability, and clock synchronization. For example, serial transmission and receiving worked perfectly from the FPGA to a PC, but required much more debugging between two FPGAs to ensure the baudbits were aligned. Finally, I learned the importance of drawing block diagrams and thinking through integration prior to writing code. Writing it out beforehand allowed me to see constraints, like those of the `sd_controller` module, VGA, and `audio_pwm` module timing constraints, to ensure that there were synchronizers in appropriate locations and allowed me to have a better sense of what was going on when trying to isolate a problem.

IV. Camera Interface (Monica)

A. Saber and Headlight Construction

We create a saber for the player using blue LEDs on a breadboard. We can power six LEDs in parallel with one 9V battery and a 68 Ohm resistor in series with each LED. We also use LEDs to identify where the player's head is. Six red LEDs and resistors on another (small) breadboard with a battery are taped to a hat for the player to wear. The LEDs are covered with tape to prevent the LEDs from over-reflecting on their surroundings, lessening the noise in the camera color detection.

B. Saber and Headlight Detection

The cameras need to detect the colors and locations of each saber. Color detection is implemented through thresholding and masking. Each camera will detect a saber by first comparing pixels to both low and high thresholds for Cr and Cb color content, and then masking threshold-satisfying pixels. The masked pixels for each are inputted in parallel into two center of mass calculation modules from Lab 4.

Originally, our goal was to have two hands (one red and one blue light on each) and use a blob detection algorithm to split the blue and red into two blobs each. The plan was to fit the masked pixels with a line of linear regression, find the perpendicular line through the center of mass, and then recalculate two center of masses for each color with the line splitting the frame. We implemented a linear regression calculation module to output b and m (as defined in Equation

1 and 2 respectively) for $y = mx + b$, the line of best fit for the masked pixels.

$$b = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2} \quad (1)$$

$$m = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2} \quad (2)$$

However, this linear regression fails when the data points fit a horizontal line - for example $x = 100$. If the line is horizontal, there is no m or b that can be inputted into $y = mx + b$ to match $x = 100$. This result was determined by displaying the line of regression on the VGA screen. When the data's best line of fit approached a vertical line, the displayed line would widely swing around a horizontal line instead. We attempted addressing this by computing the linear regression for $x = my + b$ and considering adding fixed thresholding of slope for this specific edge case. This edge case would affect the perpendicular line as well. After further efforts, this method was ultimately deemed unnecessarily complex for its application.

The proposed alternative would be to convert each pixel to HSV format, quantifying their hue, saturation, and value. This module would replace the RBG to YCrCb module. With the resulting format, we would threshold for higher saturations and values due to the bright nature of LEDs. We would then identify five different colored LEDs and mask corresponding pixels based on hue. Unfortunately, there was not enough time to implement the proposed solution, as well as limited space on the FPGA for a second hand. We instead continue forward to processing data from two cameras.

C. Camera Ranges

In order to merge the data from two cameras and obtain a 3D point in space, we first must understand the possible ranges of the camera and anticipate the potential movements of the player.

We collected sample points testing the edges of the camera's field of vision and have concluded that the camera's wider dimension can approximately capture 3 feet of width at a distance of 5 feet away from the lens. In the other dimension, the camera can approximately capture 3 feet of width at a distance of 4 feet away. The player should thus stand at least 4 feet away, and further if full arm span range of motion is desired).

As for pixel data, we store X and Y coordinates as 12 bits to not only cover the ranges below, but also to align size with the rendering and for ease of data transmission that will be discussed in the upcoming serial section.

2D camera data ranges:

- X: 0 - 274 hex (0 - 678 decimal)
- Y: 0 - 1DF hex (0 - 479 decimal)

D. Moving to 3D Space

Upon acquiring the coordinates that each camera sees an LED at, we merge their two 2D coordinates into a 3D one. We place the first camera in front of the player and the second camera looking down at the player from above so that they share one dimension. We also rotate both cameras clockwise by 90 degrees so that the wider dimension can better capture a person's arm span. Accounting for this rotation, a point (x_{3D}, y_{3D}, z_{3D}) in our 3D space would be composed of (y_2 and y_1, x_1, x_2) from the two sets of 2D camera data as depicted in Figure 1. As a result, the Z coordinate has the same range as an X coordinate would.

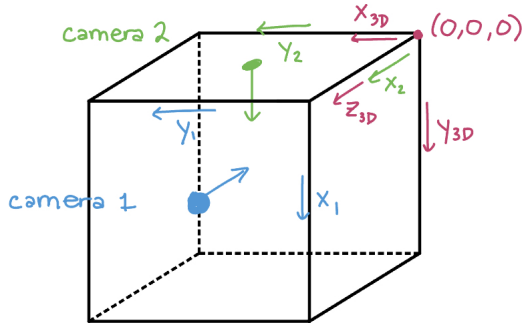

Fig. 5. 3D space constructed from two cameras

E. Serial UART Communication

In order to merge the two camera coordinates into a single 3D coordinate, we implement FPGA to FPGA serial communication discussed previously.

F. Camera Data Transmitter

The second camera, with a transmitter, continuously sends its detected center of mass X and Y coordinate for both the hand and head (even though currently only the X axis is needed to augment the Z axis, both coordinates are sent for flexibility). To continuously send bytes, once the `TxD_busy` line goes low, the `start_byte` is set high again. Since it is sending one byte after another, the receiver on the first camera needs a reference message to know which byte corresponds to which coordinate. The reference message must be impossible to recreate with camera data so the following format was designed.

The data to send includes:

- hand X coordinate (12 bits) – 0 to 274 hex
- hand Y coordinate (12 bits) – 0 to 1DF hex
- head X coordinate (12 bits) – 0 to 274 hex
- hand Y coordinate (12 bits) – 0 to 1DF hex

The transmitter cycles through the following bytes of data (omitting start and stop bits):

- byte 1 – FF (reference signal)
- byte 2 – FF (reference signal)
- byte 3 – FF (reference signal)
- byte 4 – hand X coordinate [11:4] (cannot be FF)

- byte 5 – hand Y coordinate [7:0] (can be FF)
- byte 6 – hand X coordinate [3:0], hand Y coordinate [11:8] (cannot be FF)
- byte 7 – head X coordinate [11:4] (cannot be FF)
- byte 8 – head Y coordinate [7:0] (can be FF)
- byte 9 – head X coordinate [3:0], head Y coordinate [11:8] (cannot be FF)

With this ordering, it's not possible to have three or more consecutive FF's from the data. Both X and Y coordinates can have the bottom 8 bits be FF according to our limits. But because we send the bottom 4 X bits with the top 4 Y bits, this split can no longer be FF. Additionally, we make this byte the last byte before starting over so the receiver cannot receive 4 FFs and mistake the first 3 as the reference.

These values were verified using the serial bus on an oscilloscope and comparing them to the displayed center of mass on the VGA and seven segment display.

G. Camera Data Receiver

The receiver functions with a buffer that stores the 9 most recent bytes from the transmitter. When there is a new byte at the end of a baud cycle, the previous bytes are shifted left and the new byte is stored at the lowest 8 bits. When the top 3 bytes are equal to FFFFFF, the receiver stores the other bytes as the data for the XY coordinates. Since our camera is running on a faster clock than the serial, we are able to store the correct data on the next camera clock cycle. The received X coordinates are then tied to the 3D Z coordinates.

Successful transmission of data was verified using the ILA probes on the buffer and coordinate logics and comparing them to the transmitter verification tools.

The final X, Y, and Z coordinate logics are only updated when the new information is processed and fully ready, so that only valid numbers are passed on to the rendering.

H. Insights

From this project I've learned a few valuable lessons. The first main lesson is that when it comes to implementation ideas, it is best to think thoroughly about all potential edge cases, implementation options, and resources. I brainstormed and researched various blob detection algorithms, without even considering simpler solutions that bypassed algorithms completely (such as HSV). The algorithm ended up being more complex than expected due to edge cases I had failed to consider beforehand. While the color detection algorithm was interesting to think about and learn from, I do wish I could've had more time to try the HSV route. Another insight I've had is that the ILA is very useful for debugging. However, since it takes a substantial amount of time to build for each change, it is important to figure out a bug fully and attempt a fix before running another build in order to be time efficient. And finally, I've learned to take nothing for granted in debugging. Small mistakes can be hard to find yet have great effects on functionality. I learned to methodically dig for the root of an issue.

V. 3D Renderer Raycasting (Darren)

The 3D renderer uses a raycasting algorithm in order to render blocks in a 3D landscape to the player, much like how it is done in the actual Beat Saber game. Below, I outline the modules and IP required to make a raycasting algorithm in the FPGA function correctly. Essentially, the process that was used to create this huge set of modules from scratch involved the following steps: first, a working raycasting algorithm was implemented using Python, which proved as a proof-of-concept and the point at which the FPGA implementation will build upon. Second, relevant modules that would allow raycasting to work in an FPGA were found, which involved importing 10 different 32-bit floating-point modules from the Vivado IP. Third, the Python code had to be structured in such a way that the code was compatible with FPGA code. This meant the following: since modules immediately output the result of calculations when it is done processing the input data, it was necessary to ensure that the input output pipelines of modules were synced up, and if possible, placed adjacent to each other so that the pipelining needed was minimal. This meant that the Python code had to be split up into stages, where one stage would use the outputs of the last stage as inputs, while also pipelining the relevant logic from previous stages when necessary to be input at the exact same time. For instance, in the `get_pixel_color` module, the module had to be split up into 8 stages of varying cycle delay in order to get from a block and ray information to the output pixel RGB. Many modules in this section had to be precisely pipelined in order to get correct results, since block data and XY positions were passed in every cycle and they changed very quickly to the clock cycle itself.

Finally, to ensure that the modules worked correctly, it was imperative that all modules were separately tested in simulation for correctness, since floating-point calculations were virtually impossible to test in hardware (additionally, build times using floating-point IP quickly exhausted from 10 to 30 minutes of each run). This meant that simulation modules were created for each module below, as well as integration modules to ensure pipelining correctness between modules as well. When all modules worked in simulation, the modules were tested and debugged on the hardware level only when necessary.

A. Float IP

There are 10 floating-point modules used in the development of this raycasting algorithm. This includes the operations for add, subtract, multiply, divide, less than, less than or equal, equal, reciprocal square root, and finally two modules to convert to and from a float to a signed 32-bit integer. These modules were all customized to be non-blocking, which meant that the raycasting had to be pipelined to ensure speed and correctness. Additionally, 32-bit float modules were chosen to ensure correctness. Finally, each module incurred a different delay before outputting the data. These values were carefully noted to initially predict and test pipelining logics for various logic from upstream modules.

B. 3D Vector Abstraction

In order to effectively use the floating-point IP, it was necessary to create a 3D vector abstraction for the operations needed to implement raycasting. This involved creating 12 modules for this purpose, which are the following: vector reciprocal square root, scale by a constant, add, subtract, multiply, divide, dot product, normalize, less than, compare, max, and min. The compare module would take in two 3D vectors, and based on the result of a less than vector module, output a new 3D vector that outputs each respective vector XYZ that matched the comparison condition. These modules were built and tested individually, before being used for the below modules.

A brief overview of how the raycasting algorithm was implemented is as follows: firstly, the three-dimensional block selector module must take in the 12 candidate blocks from the game state section described above, and figure out for a specific XY screen position which block is being rendered at that pixel. To do that, we will first define some terminology that is commonly used for 3D projection algorithms. Firstly, note that we have two coordinate systems: we have the XY screen pixel positions that define what is being shown on the screen, but we also have the XYZ space that defines where everything exists that is being rendered. The blocks will live in this XYZ space and be translated to the XY position to be viewed. We will have a camera, which is the location at which the 3D render is being seen. In our final project, this is considered the head position, which can change by moving your head. This is also placed in the XYZ space. Thirdly, there exist three lights in the XYZ space that allow the player to see anything in the 3D space. Two lights are placed at static locations, and the third light is placed on the saber itself.

With the terminology defined, we can begin to understand the algorithm used in this final project. In the block selector, we will begin by projecting a ray from the camera position in the 3D space to every XY pixel position, represented as a projection matrix in the 3D space that lies in front of the camera. See diagram for a visual understanding of this translation. Next, we must determine if this ray intersects with any blocks in the XYZ space. To do this, we can loop all 12 blocks currently cached and determine if any block collides with this ray. If so, we take the closest block and return this block as the correct block for the input XY pixel position.

The second part of the modules involves actually figuring out the color of the pixel that is rendered. This is done via two main modules. Firstly, we check if this ray lies on the face of the block closest to the camera. If this is true, we can do math in the 3D space to figure out whether we should render an arrow that represents the arrow of direction the saber should move to successfully "hit" the block. If this is not true, then we will use a lighting process called lambert diffusion, which essentially uses the three lights to generate the correct RGB value for this XY pixel position. Essentially, this diffusion will use the positions of each light to increase the RGB values of the pixel, where each light contributes some amount of RGB to the block pixel based on its distance to the light. This is used as our output pixel, after some constant processing, to be sent finally to the three-dimensional renderer.

The final part of our modules is the three-dimensional renderer, which simply takes in this pixel from the above module and stores it into a framebuffer of 512x384. The reason a framebuffer exists is because the float modules above take many hundreds of cycles to calculate the XY position of one particular pixel, since there are many calculations going on in the background to allow this to work. However, since the VGA module requires a pixel to be output every cycle, the solution that was decided was to always output the pixels from the framebuffer (scaled up 2x to fit the screen), and update the framebuffer when the data became available.

VI. Three-Dimensional Block Selector

A. Get Intersecting Block

The module will take in block data, such as its position in 3D space, and determine whether this block is visible, and if so, output the ray vector and t value of the block intersection, which is described below. The three-dimensional block selector will essentially run this module once for every block that is in the cache (which is 12), and then determine afterwards which is the closest block that is visible to be used downstream in the get pixel module. This is a simple check, because we know that the smallest block index is the block that is closest, since we assume all blocks come in order of time and at a constant speed. An exception to this check is that if the saber is visible at a pixel, we will always render the saber over any block.

B. Eye to Pixel

The eye to pixel module takes in a XY pixel position and, as described in the overview section, it will output a ray that maps from the current head position to the pixel. The pixel is represented as a projection matrix in 3D space, which is invisible but placed in front of the camera between the camera and the blocks.

C. Does Ray Block Intersect

The module will take in the ray from the eye to pixel module and a specific block XYZ position to determine whether the block intersects with the ray. If this is true, we output a t value, which can be used with the ray to determine where on the block the block intersects with the ray. To successfully compute this module much like every other module in this section, we use many vector modules to compute 3D vector math and determine ray-cube intersections. This module is also used to figure out where the saber should render.

VII. Get Pixel Formatted

A. Should Draw Arrow

This module will take in a block XYZ and ray XYZ position, scaled using t, to determine whether the arrow should be rendered at a particular XY pixel. This is essentially calculated using the following idea: we will only render the

arrow when we are at the face of the cube that is closest to the camera. Additionally, we can use slopes to figure out how to render the arrow based on the position of the block and the position at which the ray hits the block. This allows us to calculate the final boolean value.

B. Get Pixel Color

This module is one of the most complex to translate to SystemVerilog, as it requires a 8 stage pipeline with multiple branches that require exact pipelining to result in correct answers. Furthermore, since this module determines whether or not the pixel is rendered correctly, it was tested very extensively in simulation to verify that it works before it could be deployed to hardware. Essentially, the pixel color is determined via looping through all the lights that exist in the 3D space, of which there are three, and then calculating how much RGB each light contributes to a block position in order to generate a shading effect as seen in the game. The module will output a rgb value, which is scaled and clipped to 12 bits to be rendered on the VGA.

C. Three-Dimensional Renderer

The three-dimensional renderer module uses a framebuffer of 512x384 in order to render on the VGA. Essentially, it will take in simultaneous XY pixel positions from both the VGA and the raycasting algorithm. The module will always output XY pixel colors from the framebuffer to the VGA, and using the read-write BRAM module, it can also write to the framebuffer from the raycasting algorithm as it gets updated. This mitigates pipelining issues between the time-intensive float modules and the actual rendering of the game.

VIII. Insights

On the raycasting side, although it was very time-consuming to implement raycasting through the FPGA from scratch, it was a very rewarding experience and I came out of the project with a much greater appreciation and understanding of hardware design and programming. I found that throughout the process, aside from creating simulation pipelines to extensively test everything before pushing to hardware, having a close software implementation copy of the code, especially in Python, helped a lot with debugging. It was a lot faster to verify design changes in software than it was in hardware, which definitely made it possible to complete this project on time for the end of the semester.

IX. Evaluation of Design

For our final FPGA that holds all game state and rendering logic, we end up using about 72% of the Slice LUTs, 89% of BRAM, and 88% of the DSPs available in the FPGA. Most of the usage comes from the extensive floating-point modules, vector operations, and complex pipelining that occurs in the raycasting algorithm. At one point in the design, the modules actually took up over 100% of the DSPs available. To resolve this, LUTs were used in place of DSPs for some floating-point modules, and logic was folded when possible instead of instantiating many expensive vector/float modules in parallel. For instance, originally the does ray block intersect module was done in parallel for the entire cache at once, but this led to having a usage of nearly 175% of the existing DSPs. The final design runs the module once for every block in the cache and once for the saber as a state machine in order to use less modules and output once.

The latency and throughput of the modules written are documented, since they are required to correctly compute the pixels from the raycasting algorithm. This is in the order of a few hundred to a thousand cycles, which is more than fast enough to successfully render onto the screen without any noticeable lags. The entire set of modules fits timing requirements, with a slim WNS of 0.238.

In our checkoff list described in the initial project report, we aimed to be able to play a song, and track using the camera in a 2D space. In our goal, we aimed to use an SD card to play music, communicate between FGPAs, and pipeline camera data. In our stretch goal, we aimed to implement raycasting, head tracking, and custom songs. We have reached goals in each of these sections. Although we did not have sufficient time to integrate all the items in the checkoffs we defined, we believe we have implemented a well-implemented game that achieves all the main goals that are required to make our game work: music playback, raycasting, head tracking, and saber tracking, as well as a high score to be shared after playthrough. On the raycasting side, since the entire raycasting algorithm had been implemented from scratch, it is entirely possible to minimally change the design in order to add more objects into the 3D space, for example. Additionally, since we have the entire song pipeline written out, we can easily switch out the song and generate new blocks for the new song. Finally, the camera data can be augmented minimally by incorporating the Z coordinate information to make tracking more immersive.

X. Block Diagrams

Block diagrams begin on the following page.

XI. Code Repositories

https://github.com/shreyka/huah_cubed
https://github.mit.edu/skarpoor/music_111_fp.git

Contributions and Acknowledgment

Shreya was in charge of the music jukebox in the project. She implemented the audio pipeline to play a speaker output from the sd card, and also play the 'hit' sounds when a block is hit in gameplay. In addition, she wrote the 2 way UART serial communication and used it to send serial data from the game play FPGA transmitter to the music receiver.

Monica implemented the camera interface and created the hand/head LED set. She extended Lab 4 code for player head and hand detection. In addition, she debugged serial communication between two FPGAs and utilized serial to send camera data and create 3D coordinates from two sets of 2D coordinates.

Darren was in charge of implementing the rendering pipeline and game logic. He created the 3D rendering algorithm inspired by a raycasting algorithm from another class (6.172), rewrote the algorithm in Python, and tuned it so that it could be used for Beat Saber and also be SystemVerilog-compatible, before implementing the entire pipeline through SystemVerilog.

Thanks to Fischer, Jay, and Joe for an amazing semester of learning, much help with debugging, and for the great knowledge!

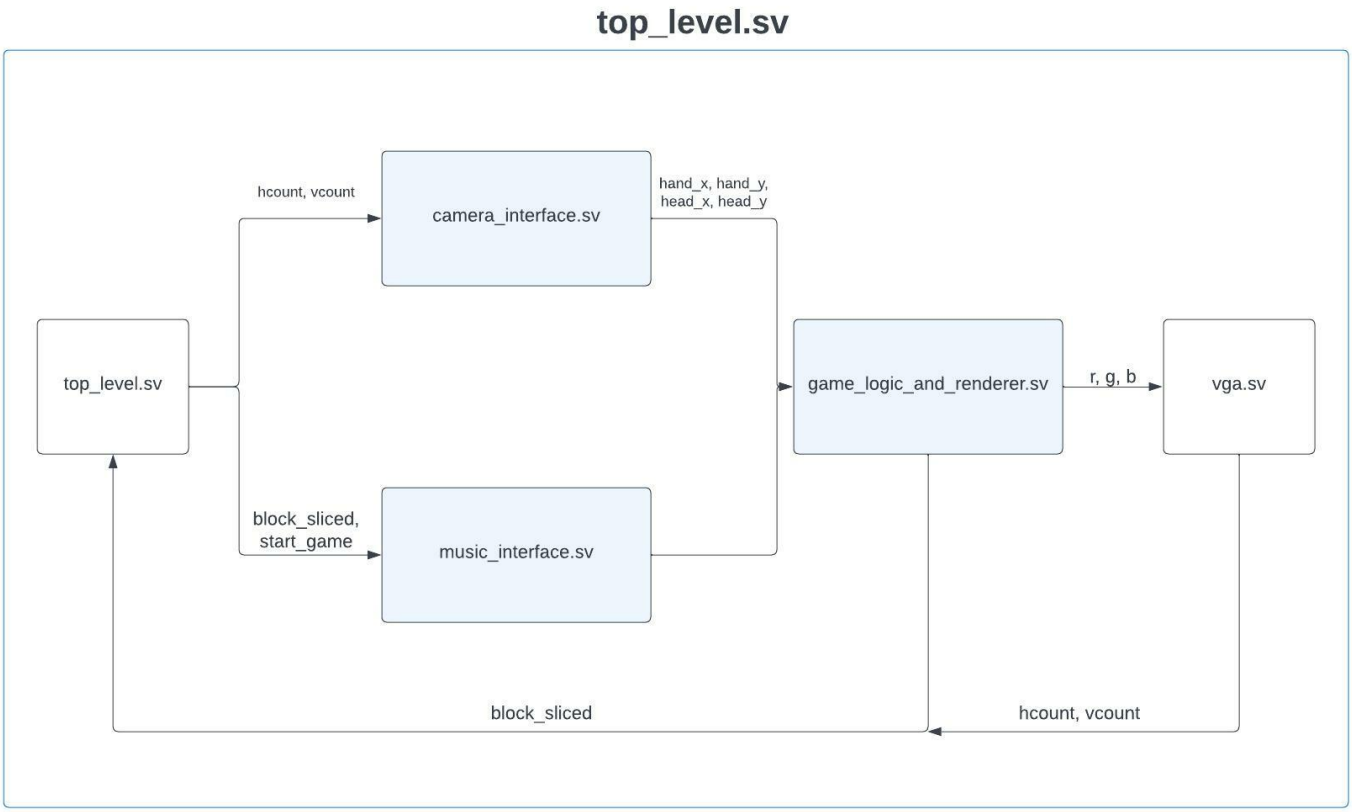

Fig. 6. Top level block diagram

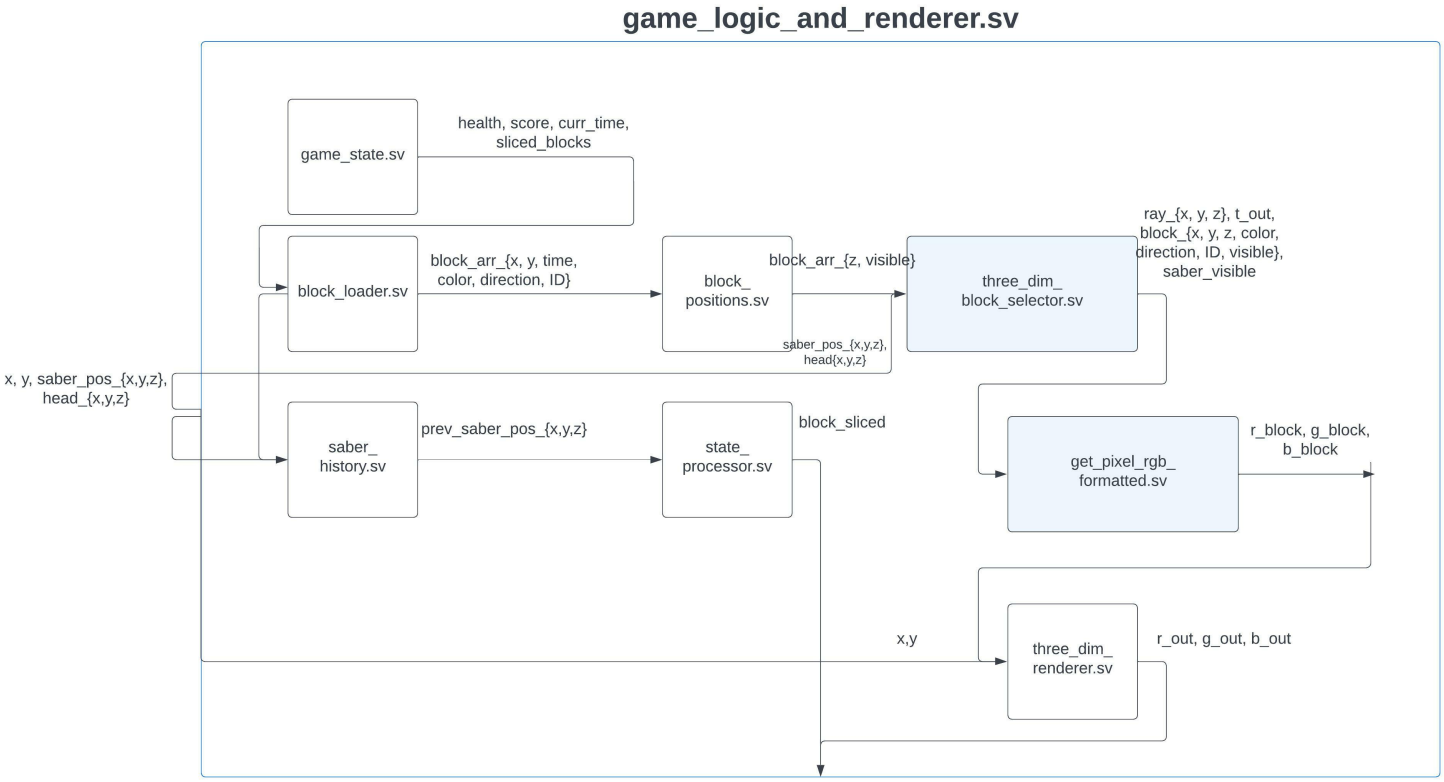

Fig. 7. Game logic and renderer modules

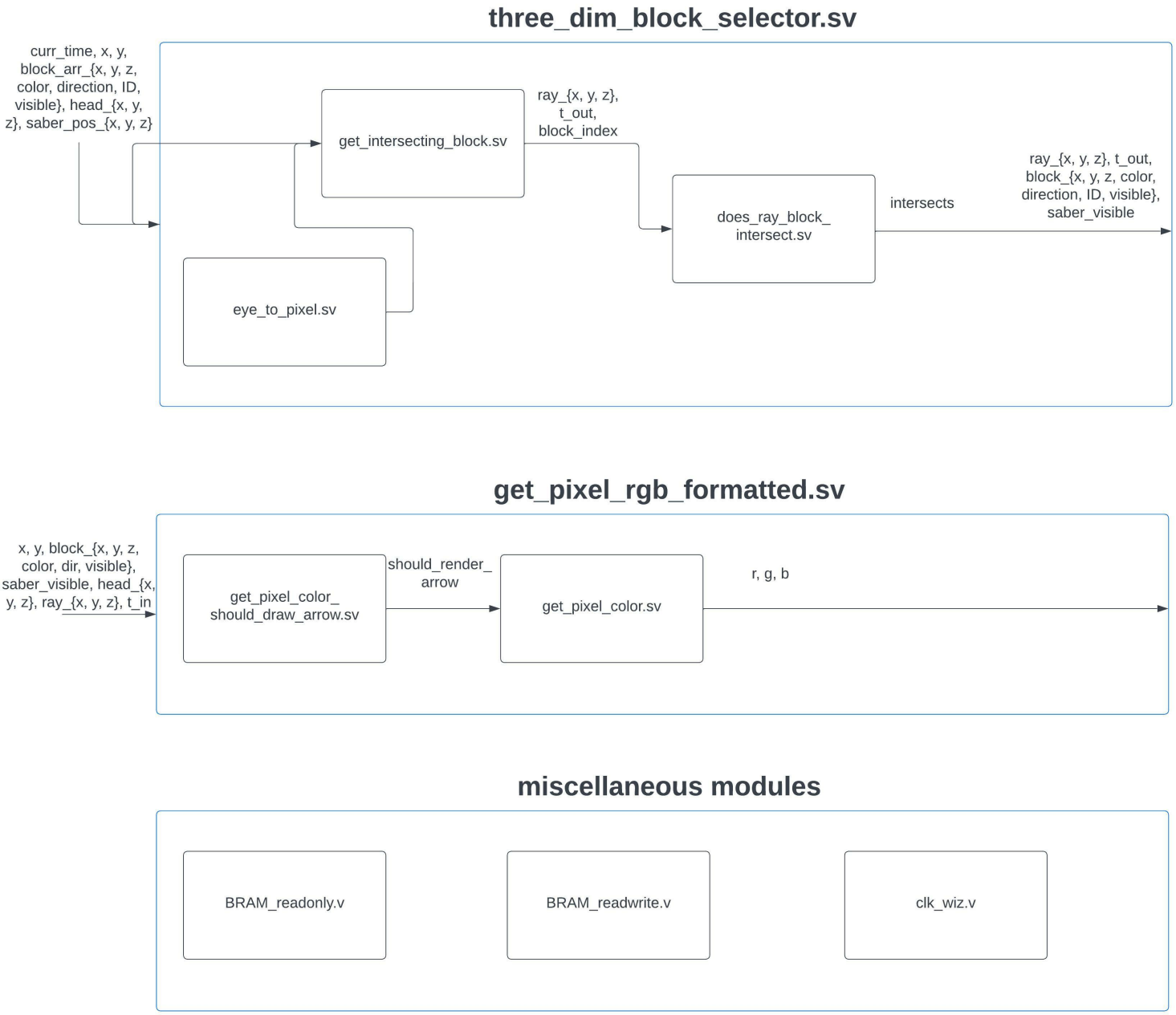

Fig. 8. Submodules used in the raycasting module

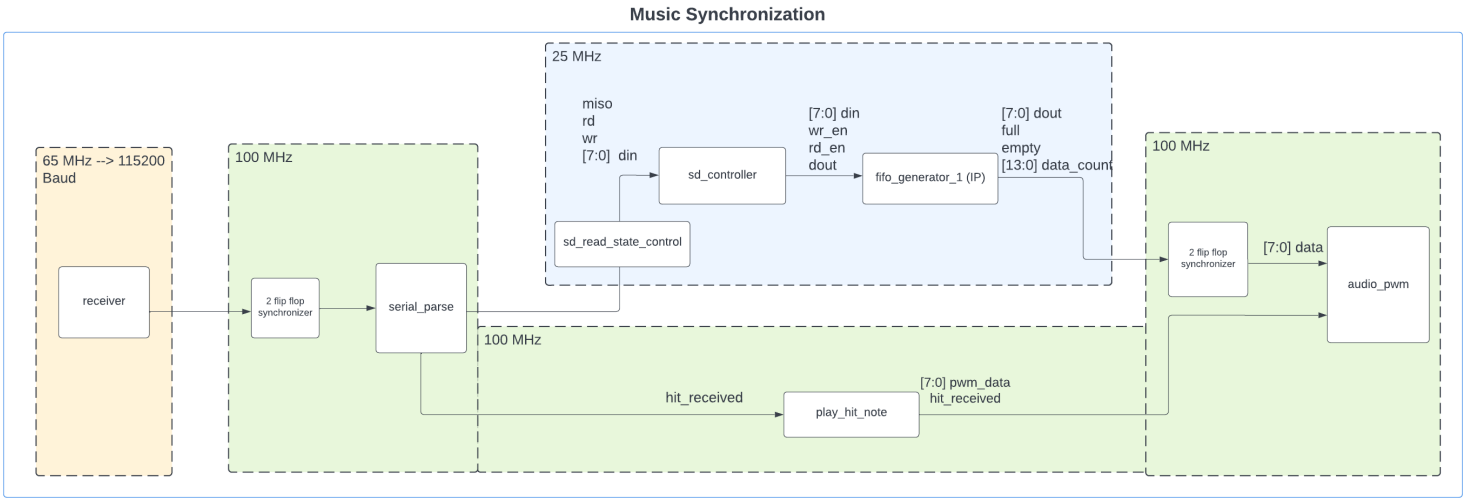

Fig. 9. Music Synchronization Block Diagram

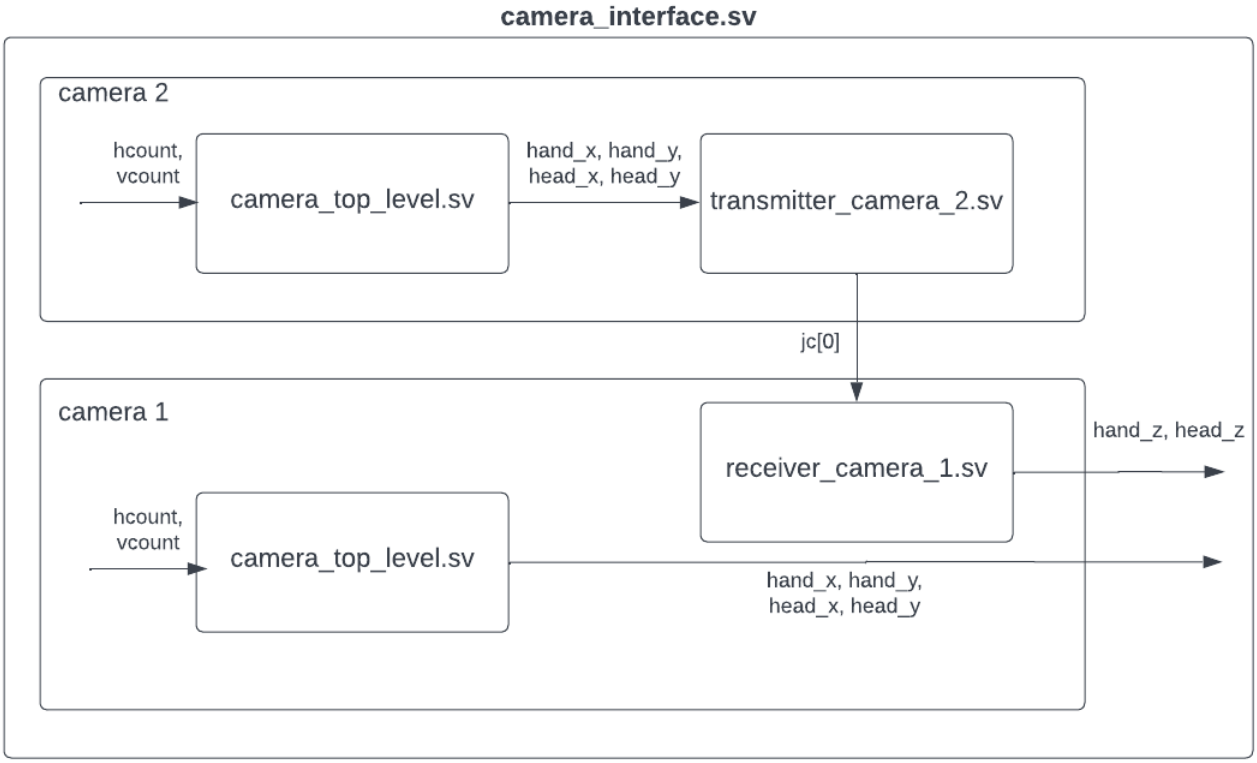

Fig. 10. Camera interface block diagram

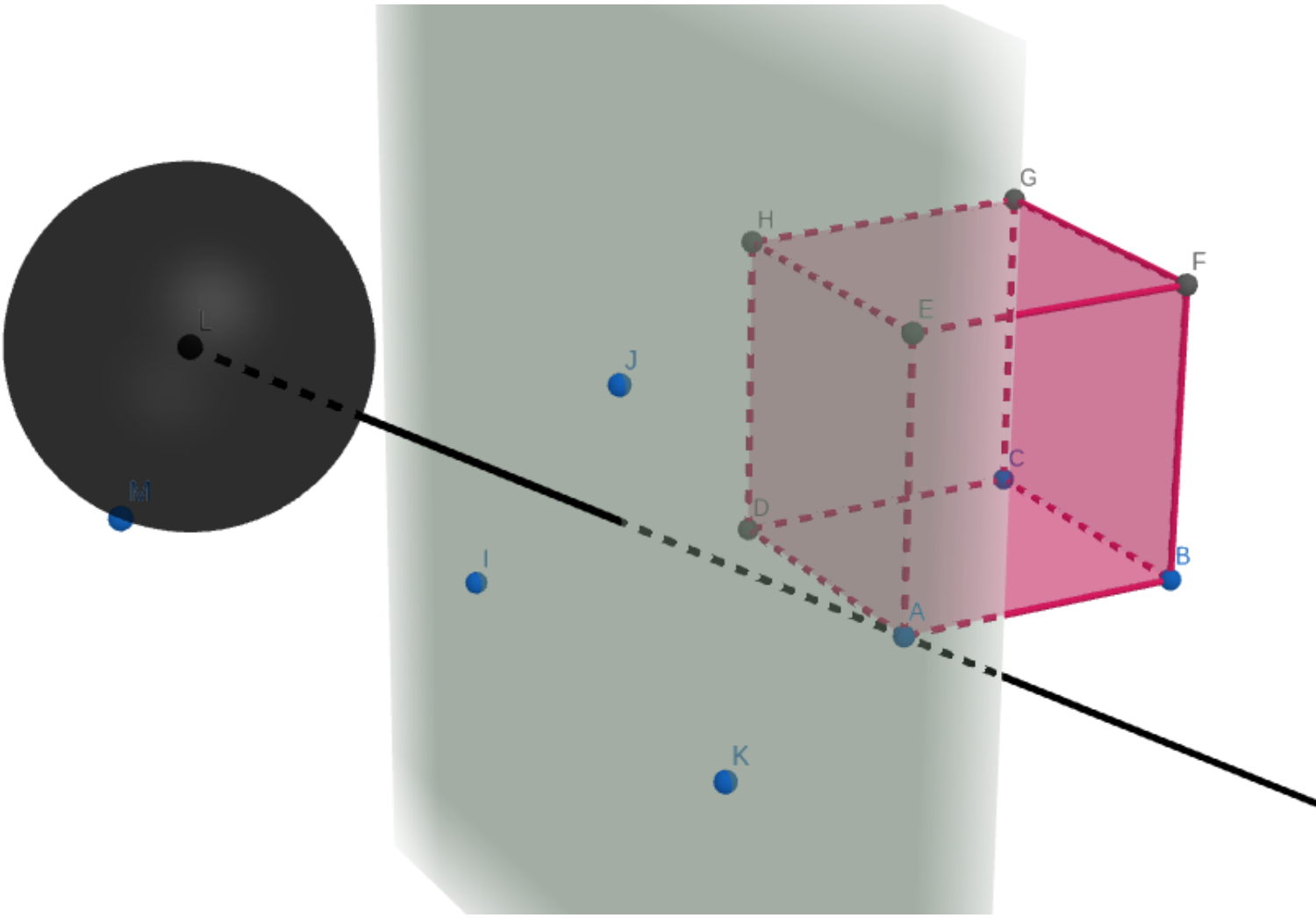

Fig. 11. Raycasting objects

Fig. 12. Float Functions

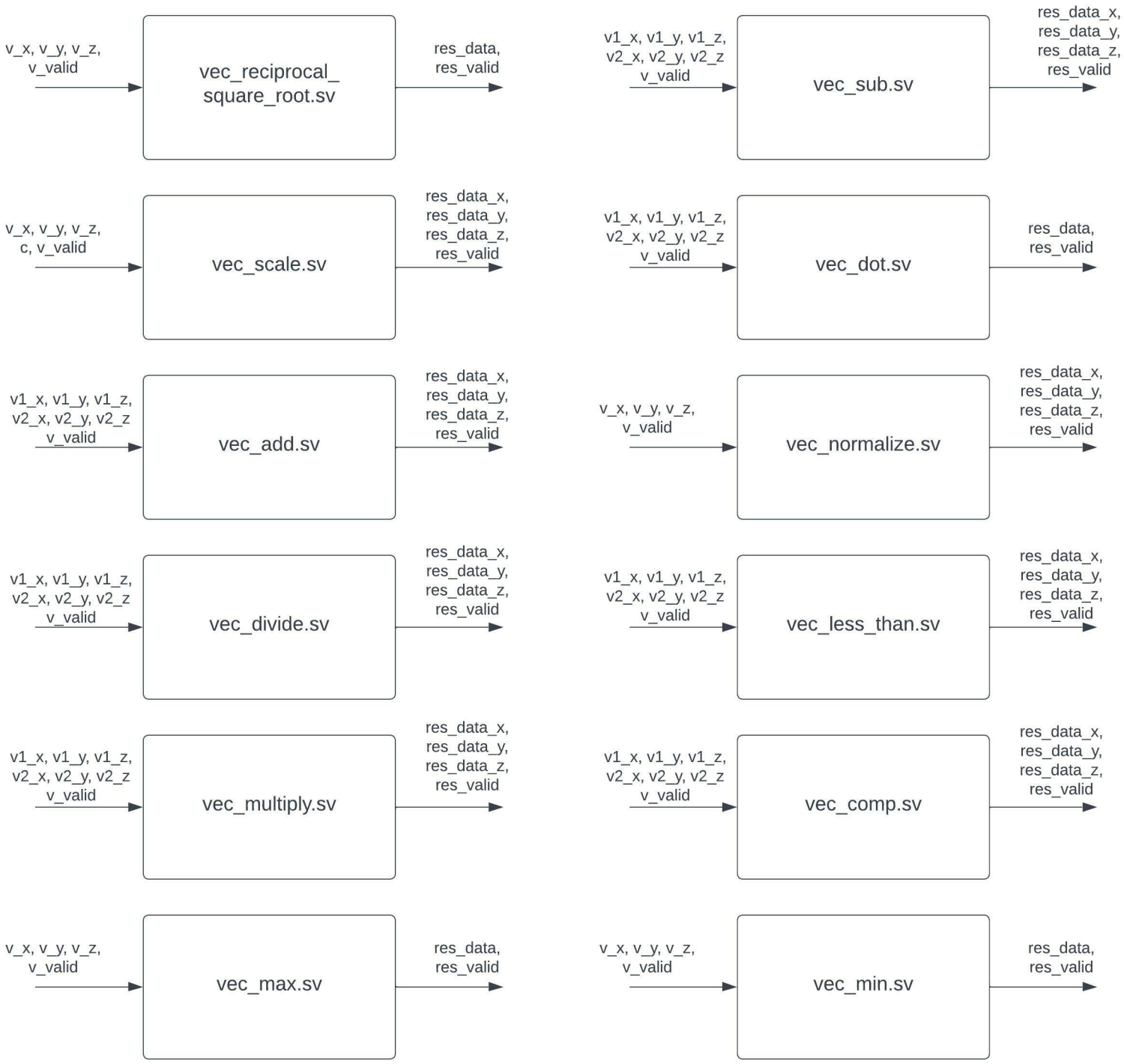

Fig. 13. All vector function modules used for raycasting