

Finger-Propelled Graphics Assistant

Final Report

1st Roshni Parulekar-Martins

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
roshnipm@mit.edu

2nd Fiona Lu

Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Cambridge, MA, USA
frofit@mit.edu

Abstract—In this project, we developed a hand-guided real-time 3D model visualization tool. Using a camera and gesture tracking system, users can perform intuitive actions such as rotating along different axes, and translating 3D objects. The system will implement point cloud projections to render 3D models into 2D coordinates for display on an HDMI screen. These 2D projections will then be dynamically manipulated based on the user's hand movements, allowing for natural and responsive interaction without relying on conventional input devices like a mouse or touchpad. We also implement a UART transmission system for sending points for a new point cloud model a user may want to render to the screen.

Index Terms—Digital systems, Field programmable gate arrays, cameras, 720p HDMI/VGA displays, UART transmission.

I. INTRODUCTION

In this project, we developed a **hand-guided real-time 3D model visualization tool**. Using chroma-based center of mass tracking, users are able to rotate point cloud models along different axes and translate them. These coordinates are then projected onto to a 2D plane for display on an HDMI screen. Key properties of our system include:

- 1) **Finger Tracking:** The FPGA can spatially cluster pink pixels based on the user moving pink foam. It then finds their centers of mass and convert these into control signals for model transformations (e.g., rotate X degrees, translate by Y points in x/y space).
- 2) **3D Point Cloud Rendering:** Implement a point cloud 2D display pipeline capable of performing rotation and translation transformations efficiently on FPGA hardware.
- 3) **UART Transmission:** We are able to transmit new points to the FPGA board from a connected computer after the build has already been flashed using a UART-based transmission method.

II. PHYSICAL CONSTRUCTION

III. FPGA HARDWARE REQUIREMENTS

- Memory
 - On-board 2.764 million-bit BRAM storage
- Input

MIT EECS 6.205

- 100 MHz system clock input
- User I/O, slide switches
- Camera Interface, including horizontal and vertical sync, and camera pixel clock input
- Output
 - HDMI Video Output Interface, including RGB Data Channels and differential pixel clock
 - Camera Interface, including external clock output to camera

IV. OV5640 DIGITAL VIDEO PORT

- Camera horizontal and vertical sync
- External reference clock
- I^2C Configuration Interface
- Pixel clock input

ALGORITHM AND SYSTEM OVERVIEW

In software, the point cloud projection pipeline has three steps:

- 1) Define the points in 3D coordinates.
- 2) Multiply by 3-by-3 matrices to enact any rotation transformations.
- 3) Multiply the x- and y-coordinates by the focal length.
- 4) Divide the x- and y-coordinates by the z-coordinate (depth) to obtain the final 2D-projected points.

To translate the algorithm into hardware, we made 3 key system design choices:

- 1) The rotation matrix along each axis was discretized into multiples of $\frac{2\pi}{512}$ radians.
- 2) The rotation matrices were scaled by 256 for fixed point calculations.
- 3) Division was pipelined with the fixed-latency, 32-bit, 16-cycle divider from lecture.

Below are four crucial modules from our design and which step in the point cloud algorithm they correspond to:

- 1) `pointInput` accesses the dual-port 2-clock BRAM storing the original 3D coordinates of all points.
- 2) `coordinateTransform` conducts the 3-by-3 signed matrix multiplication with fixed-point math. There are 3 of these in `top_level`, one for rotation along each axis.

- 3) `perspectiveProjection` does the 16-cycle division and handles signed portions of the 2D projection step.
- 4) `model_2d_buffer` manages the BRAMs used to store the 2D-projected points in a frame buffer for writing to the screen.

In addition, we use the `com_to_transform` module to convert the center of mass signals from `center_of_mass` (implemented in Lab 5) to rotation matrices along each axis. We also implemented a `uart_points_receive` FSM module to collect new point coordinates from the computer to be stored on the FPGA, which works with the `uart_receive` module from Lab 3.

Finally, a `model_2d_buffer` signal controller FSM is implemented in `top_level` to ensure all transformations, projections, and point storage occur during the bottom blanking period of the HDMI video stream.

DESIGN AND IMPLEMENTATION

V. FINGER TRACKING TO TRANSFORMATION MATRICES



Fig. 1. Screen with video center of mass input pre-point cloud implementation.

A. Center of Mass (COM) Tracking

Finger tracking is done with the center of mass thresholding, similar to what was implemented in Lab 5. However, instead of keeping the entire video stream in BRAM storage, only the thresholded pixels need to be kept. Thus, the BRAM width need only be 1, saving memory storage for the 1280-by-720 point cloud frame buffer. Since this is displayed in the corner, downsampled by 4 in each dimension, $320 \times 180 = 57600$ bits of memory are actively used at a time.

How much to translate or rotate by is measured by changes in x_{com} and y_{com} . In the case of translation, the center of mass is directly added to all the points of the point cloud. In the case of rotation, the center of mass is converted to a discretized angle theta which determines the transformation matrix for changing the points.

B. COM to Transformation Matrix

The conversion from center of mass to the transformation matrix is done by the module `com_to_transform`. This module takes as input the current x_{com} and y_{com} as well as the intended `transform_type` (either x, y, or z). Thus, there are three instances of this module in `top_level`, one for rotation along each axis. To rotate points along the x-axis, changes in y_{com} are used; to rotate points along the y-axis, changes in x_{com} are used; and to rotate points along the z-axis, changes in y_{com} are used (see Fig. 2).

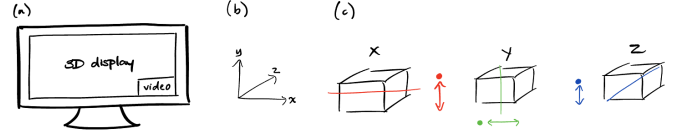


Fig. 2. (a) 3D point clouds are displayed in the main portion of the screen. (b) Axes of the point cloud. (c) X-/Y-/Z-axes and the camera X-/Y-axes along which COM changes are used to rotate them.

This module additionally contains two BRAMs: one for indexing into the `sineROM` and one for indexing into the `cosineROM`. Based on a given camera axis' center of mass, we calculate an address for indexing into each BRAM. Based on the output and the axis of rotation, the FPGA calculates one of the following three transformation matrices. Each BRAM contains 512 lines, each corresponding to $2\pi \times X/512$ radians, where X is the line number. For the purposes of fixed point math, each sine or cosine value is interpreted with a scaling factor of 1/256.

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta_x & -\sin \theta_x \\ 0 & \sin \theta_x & \cos \theta_x \end{bmatrix}$$

$$R_y(\theta_y) = \begin{bmatrix} \cos \theta_y & 0 & \sin \theta_y \\ 0 & 1 & 0 \\ -\sin \theta_y & 0 & \cos \theta_y \end{bmatrix}$$

$$R_z(\theta_z) = \begin{bmatrix} \cos \theta_z & -\sin \theta_z & 0 \\ \sin \theta_z & \cos \theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Fig. 3. Rotation matrices along each axis (x/y/z)

C. Switch Controls

The following switches are used to control the display output and transformations:

- 1) **sw[0]**: used to control whether or not the bottom-right corner video display is showing.
- 2) **sw[2]**: used to control whether x_{com} and y_{com} are controlling the center of the point cloud. After the switch turns off, this translation amount is saved and continued to be used for display.

- 3) **sw[3]**: used to control whether a rotation is being implemented or not. After this switch turns off, the rotation matrices are saved and the last-saved rotation amount continues to display.
- 4) **sw[4]**: used to control what axes of rotation are changeable at the moment. On means that the x- and y-axis rotations are being controlled (with `y_com` and `x_com`, respectively). Off means that the z-axis rotation is being controlled by `y_com`.

VI. 3D POINT CLOUD RENDERING

A. Point Input (*pointInput*)

In the case that a transformation has not yet been processed, this module reads a new point from BRAM memory. The point will contain 3 fields, x, y, and depth (z), all represented with 12 bits of precision. Each point has an associated `point_id`, which corresponds to the address of memory it is stored in. This ID will become important later on when storing the buffer coordinates.



Fig. 4. The `pointInput` storage method has x coordinates on the MSB end and z coordinates on the LSB end with each point having an associated `point_id` as the address.

Our initial point cloud testing file contained a cube with vertices at combinations of 1024 and -1024 . We tested point clouds with thousands of points from a repository with common 3D test models. Using a python script, we used K-means clustering to downsample the points from various `.obj` files to the range of thousands of points, scale their max coordinates to the range of $[-1.5, 1.5]$, and then multiply them by 1024 prior to 12-bit two's complement storage.

Finally, we want to highlight that this BRAM is a dual-port 2-clock buffer in order to accommodate the UART transmission of new points, as described later. Thus, the module also takes as input `new_point_ready`, `new_point`, and `addrd` to be able to write newly-transmitted points to memory.

B. Coordinate Transform (*coordinateTransform*)

This module multiplies the read output of `pointInput` by a 3x3 rotation matrix. One cycle is used for multiplication, and one cycle is used for addition. Notably, we gave this module a bit width of 24 to account for the scaled fixed point representations of the sine and cosine values. Following each rotation multiplication, all results are shifted down by 8 bits to account for this fixed point scaling factor of 256.

C. Perspective Projection (*perspectiveProjection*)

This module projects the output of `coordinateTransform` onto a 2D screen. It does so by dividing the x and y coordinates by the depth, using divider modules with a fixed latency of 16 cycles (`divider4.sv` from lecture). Since this module cannot

handle signed math, the perspective projection module removes sign and reintroduces it post-division.

Additionally, inputs prior to this module contain a fixed depth addition of 1310720 ($256 \times 1024 \times 5$) to ensure that the point cloud is beyond the origin and can actually be visualized on the screen. X- and y-axis inputs to the module are also multiplied by 500 pixels in order to ensure that points are projected to the screen.

Due to these multiplications and additions, as well as the usage of the pre-built `divider4` module, we kept the bit width of this module at 32. Following this module in `top_level`, the translation amount is added to the point before it gets written to the frame buffer.

D. Model 2D Buffer (*model_2d_buffer*)

This module manages a 2D frame buffer for displaying 3D point cloud projections on an HDMI display. To accomplish this, the module actually maintains two single-port BRAM memories:

- 1) **point_map_ram**: Maps `point_id` to (x, y) coordinates on the next buffer, storing the previous position of each point for efficient clearing of old pixels.
- 2) **frame_buff_ram**: A 1-bit-per-pixel frame buffer (1280×720) where 1 = white pixel (point present) and 0 = black pixel (no point).

In an inefficient version of this system, the module would have to clear every pixel of `frame_buff_ram` before writing to new points. However, this would take up a majority of the active drawing period and would necessitate the use of a ping-pong double buffer. Instead, we use one the `point_map_ram` as an addressing agent for `frame_buff_ram`. This way, we only spend $O(points)$ time clearing the `frame_buff_ram` rather than $O(screenheight \times width)$.

To accomplish this, the module takes in a key input in addition to the `point_id`, projected x-, and projected y-coordinates: `clearing_phase`. This input determines whether upon valid input the module (1) reads `point_id`'s address from `point_map_ram` and (2) writes 0 to that address in `frame_buff_ram`; or (1) writes `point_id`'s new address to `point_map_ram` and (2) writes 1 to that address in `frame_buff_ram` (please see the next section and Fig 5 for more clarification).

The module also takes in the input `point_off_screen` to ensure that if a point goes outside of the 1280×720 viewing window, nothing will be written to `frame_buff_ram`.

E. Model 2D Buffer Controller FSM

We accomplish all our point cloud calculations for the next frame during the bottom blanking period of the HDMI video output. To do this, we use an FSM in `top_level` to manage the clearing and recalculation steps of the point cloud pipeline.

The following states are employed in the controller FSM:

- 1) **IDLE**: The FSM stays in this phase while the points are actively rendering to the screen and after the FSM

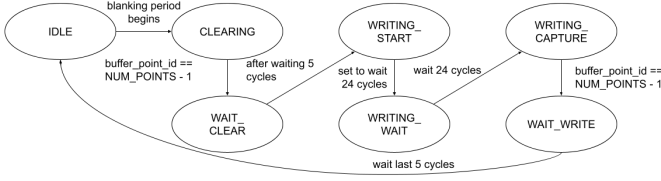


Fig. 5. model_2d_buffer controller FSM in top_level

has gone through all states during the bottom blanking period.

- 2) **CLEARING:** Right after the HDMI enters the bottom blanking period, the FSM moves into this phase. During this phase, the `clearing_phase` signal to `model_2d_buffer` is high and we begin a counter for the `point_id` signal to iterate through all points and clear their positions in `frame_buff_ram` to 0. This stage takes a total of `NUM_POINTS` cycles. Once we reach the last point, the FSM transitions to `WAIT_CLEAR`.
- 3) **WAIT_CLEAR:** While the throughput of our point cloud pipeline is 1, we wait 5 cycles during this stage at the end to let the last point finish going through the latency of `model_2d_buffer`. Then, we move to `WRITING_START` to begin writing the new point cloud points to the frame buffer.
- 4) **WRITING_START:** This phase sets the cycle counter to 24 (the latency of `pointInput`, 3 iterations of `coordinateTransform`, and `perspectiveProjection` for `WRITING_WAIT` to go through).
- 5) **WRITING_WAIT:** This is a waiting stage to make sure the first point full goes through the point cloud pipeline finishes all computations before moving onto `WRITING_CAPTURE`.
- 6) **WRITING_CAPTURE:** During this phase, we once again begin a counter for `point_id` to iterate through all `NUM_POINTS`, except this time the `pointInput` module receives a valid signal to begin the 3D transformation and 2D projection computations. This phase takes a total of `NUM_POINTS` cycles. At the end the pipeline sets the wait counter to 5 cycles for `WAIT_WRITE`.
- 7) **WAIT_WRITE:** This is the final waiting state for the last point to finish going through `model_2d_buffer`.

This means that we use a total of $X + 5 + 24 + X + 5 = 2X + 34$ cycles in the bottom blanking period to do point cloud calculations and write them to `frame_buff_ram`. Given that our bottom blanking period is $1650 \times 30 = 49500$ cycles long, we decided to test if our design could handle $\frac{49500 - 34}{2} = 24733$ rounded down to 24000 points. Unfortunately, the high-performance BRAMs were unable to handle this many points during build and allocated 86 RAMB36 blocks. However, this design still allows our system to successfully render and

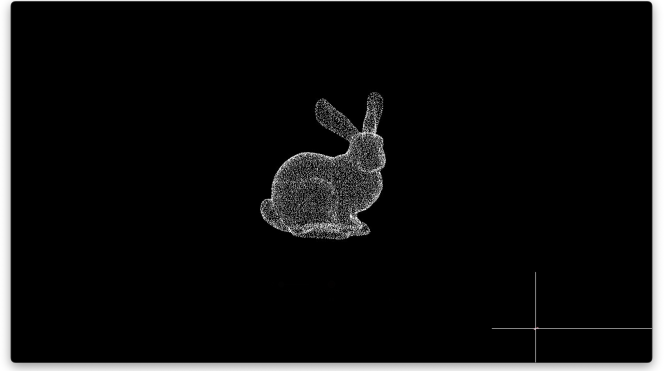


Fig. 6. Our final working design with 16,000 points rendered to the screen.

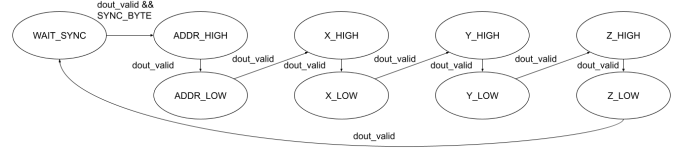


Fig. 7. UART FSM structure and transitions to receive new points from the computer.

transform up to 16000 points (see Fig. 6). Further discussion on this is in the Resource Utilization section of Evaluation.

F. UART Transmission of Points

We made progress toward our stretch goal of being able to transmit points from the computer to the FPGA board using a UART transmission protocol. We connected the `uart_receive` module from Lab 3 for receiving points to another FSM module `uart_points_receive` for collecting the line to store in memory for each point.

This FSM has 9 stages in total. First, it expects to receive a syncing byte of `8'hAA`, which we assumed was unlikely for the UART to otherwise receive during the `WAIT_SYNC` stage. Following that, it transitions into accepting the new point's address, followed by its x, y, and z-coordinates. Each state transition after `WAIT_SYNC` is completed using the `dout_valid` signal from the `uart_receive` module.

Our address was at most $\lceil \log_2(16000) \rceil = 14$, and each coordinate had 12 bytes. Since UART protocols send one byte at a time, we modified the sending of our point data based on bit width. We divided our address bits into high and low portions of 8 bits each, and divided our axis coordinates into a high portion (with 4 bits of 0 padding and the upper 4 bits of the coordinate position) and the low portion (all 8 lower bits of the coordinate position).

VII. CHECKLIST

We were able to accomplish the commitment goals from our checklist. This consisted of a working 1-bit width camera display, the use of separate x/y/z switches for rotating along different axes, using 1 pixel per point, and using BRAM-only point cloud storage with 0/1 black and white per point in the

cloud displayed to the 2D screen. We also tested our system with various numbers of points, and our system was able to handle up to 16000 points for display.

We were also able to accomplish one of our goals: replacing two of the x-axis and y-axis control switches with just one switch to control the x- and y-axis rotations.

Unfortunately, we were not able to successfully integrate DRAM into our project. Originally, our goal was to replace the second BRAM of `model_2d_buffer` with a full-color RGB buffer and include color values with our 12-bit precision x-, y-, and z-coordinates. This would have also enabled us to accomplish our second goal of recoloring points by depth before rendering to the screen.

Since our DRAM goals were difficult to integrate into the project later on, we decided to implement our stretch goal of opening a port for transmitting new points on the FPGA instead. While we did not have the chance to implement the webpage portion of this goal, the hardware portion is fully functional and would only require the coding of some HTML files in software, which is not within the scope of this class.

VIII. EVALUATION

A. Latency and Throughput

The worst negative slack (WNS) of our design is more or less 0.370ns, meaning that our design's total negative slack (TNS) is 0ns and meets timing constraints.

The latency of our point cloud pipeline is divided up as follows:

- 1) **pointInput:** This single-port BRAM block has a latency of 2 cycles.
- 2) **coordinateTransform:** There are 3 instances of these executing computations one after the other. Each one takes 2 cycles (one for multiplication and one for addition), leading to a total latency of 6 cycles.
- 3) **perspectiveProjection:** Each parallel 32-bit divider module has a latency of 16 cycles.
- 4) **model_2d_buffer:** 5 cycles total: (1) the first BRAM takes 2 cycles to read/write from; (2) it takes one cycle to do multiplication for calculating the appropriate address in the next BRAM; (3) it takes 2 cycles to read/write from the second BRAM.

This leads to a total point cloud pipeline latency of $2 + 6 + 16 + 5 = 29$ cycles.

B. Resource Utilization

Altogether, for 16,000 points we around the following amount of resources:

- 1) **LUTs:** 4275/32600 (13.11%)
- 2) **Slice Registers:** 143/65200 (0.22%)
- 3) **DSP Blocks:** 34/120 (28.33%)
- 4) **RAMB36 Tiles:** $(56 + 6/2)/75$ (78.67%)

But when we try to use 17,000 points, the number of RAMB36 tiles used during build jumps to 86, which does not fit on the FPGA. This does not align with our calculations. Each point takes up 36 bits of storage in `pointInput`

and 21 bits of storage in `point_map_bram`, leading to $(36 \times 1000) + (21 \times 1000) = 36000 + 21000 = 57000$ more bits to be stored in BRAM. Even if we consider the BRAMs rounding up, this should theoretically only take up two more RAMB36 tiles on the board, not 27 more. We suspect that at 16,000 we reach some sort of placement limit that causes fragmentation of other BRAMs on the board.

IX. REFLECTION

Overall, we are proud of the fact that our system is able to handle the recalculation and projection of 16,000 points onto the screen within the span of the bottom blanking period. However, we wish we had done more experiments earlier on to understand why more BRAM tiles were being allocated than expected so that we could test our theoretical limit of 24,733 points, which should theoretically be feasible in memory.

Secondly, we miscalculated the amount of work it would take to integrate DRAM into our project after already making it work with BRAM. Our modules were very specifically pipelined based on the 2-cycle BRAM latency delay, and trying to change this type of memory to DRAM later included complexities we had not originally accounted for due to the unfixed latency nature of this type of memory. Additionally, we would have had to reconfigure the setup from lab 6 since we would not be sending 128 bits at a time, and would want the `ddr3_controller` to only focus on certain bytes of information from the signal. If we were to do this project again, we would have started off by using DRAM instead of starting off with BRAM and later on trying to switch to DRAM so that we would not have to significantly rework our system. This would have made it much easier to reach our goals of coloring and shading later on.

Finally, one thing we initially planned for was point write-back post-coordinate transformation so that we could create our transformation matrices in `com_to_transform` based on the *change* in the location of the center of mass over time rather than the *absolute* location of the center of mass. This means that with our current implementation, our transformations all lie along the point cloud's original x- and y-axes rather than ones recalculated based on previous rotations. However, when we tried to implement the writeback of points, we ran into issues with rounding due to the consistent shifting of bits up and down by 8 in our fixed-point calculations, leading to our point clouds successively getting smaller and smaller with every rotation. As a future step, we would implement rounding at every bit-shifting step to prevent the gradual shrinking of the point cloud.

X. CONTRIBUTIONS

Roshni worked on the camera input feed, `uart_points_receive`, `model_2d_buffer`, its controller FSM, and the higher-level system design and integration onto the FPGA. Fiona worked on the `com_to_transform` and point projection modules.

XI. ACKNOWLEDGEMENTS

We would like to thank Joe Steinmeyer for a wonderful semester of this class. We have truly learned a lot about digital design, and especially loved the units on video (prompting our use of it in this project). We would also like to thank our TA, Kiran, for helping us so much in lab and giving us invaluable system design suggestions.

XII. HIGH-LEVEL BLOCK DIAGRAMS

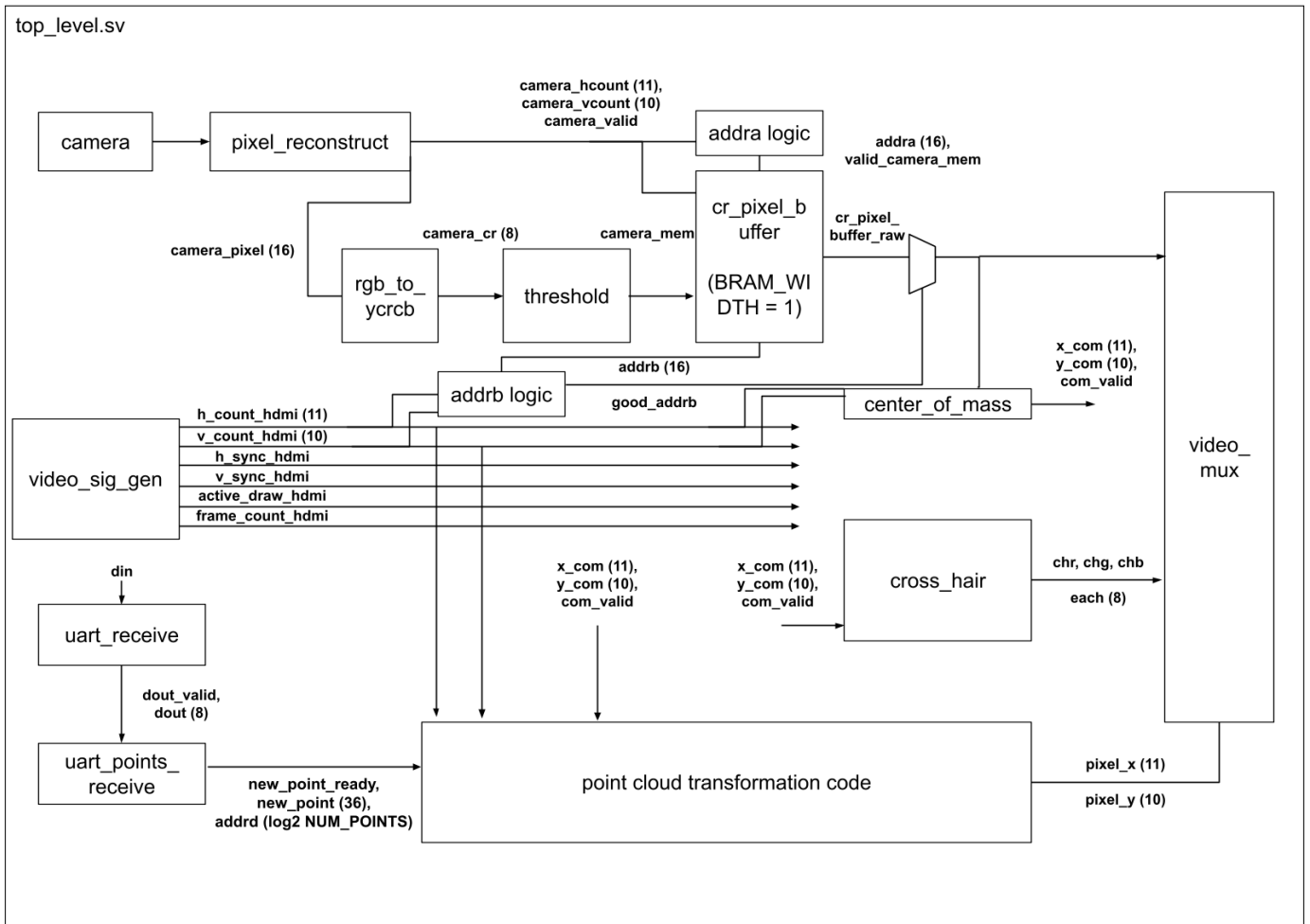


Fig. 8. Video Center of Mass Input Block Diagram

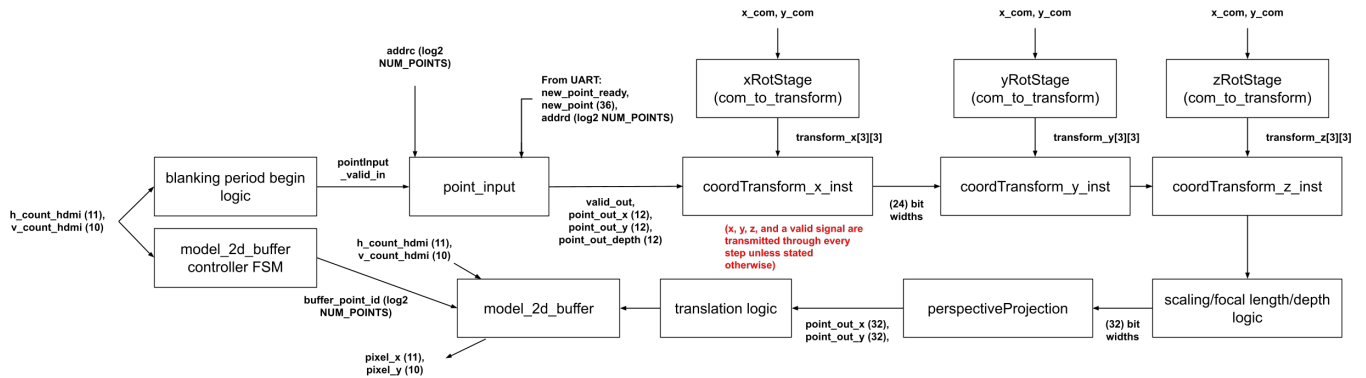


Fig. 9. Point Transformation Pipeline Diagram