AR Crayon Physics

1st Younghun Roh
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
yhunroh@mit.edu

2nd Disha Kohli
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
dishakl@mit.edu

3rd Daniel Vargas
Department of EECS
Massachusetts Institute of Technology
Cambridge, MA, USA
davarg5@mit.edu

***Abstract*—We present Augmented Reality (AR) Crayon Physics implemented with FPGA. Crayon physics is a game that allows users to draw objects which follows the laws of physics, and interact with the existing objects to move a ball into the goal. In this project, we add an augmented reality aspect as well as a user-directed sandbox, and discuss possible extensions such as the inclusion of a multiplayer mode.**

Instead of conventional touchscreen or mouse-driven input, a camera will be used to capture players' gestures as they draw objects with their fingers in the air. If time permits, we will aim to allow user gestures to directly manipulate the sprite (i.e. flicking, pushing, etc) via physical sensors and actuators.

We plan to explore different gameplay modes, starting with a sandbox mode, and expanding to include pre-built levels, and multiplayer mode if feasible.

This project will be divided into three distinct components– the physics engine, which will compute the effects of gravity and object collisions, followed by the game logic finite state machine and user experience design, rounded out by the finger tracking for user interaction in the sandbox and levels.

I. Sandbox Logic (Disha)

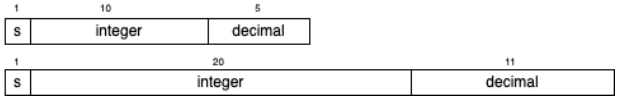
The sandbox allows users to place fixed and movable objects on the screen that the ball can interact with when the simulation commences. Most of the time, the program is in an idle state, when the user is not actively interacting with it. However, if a user toggles the two right-most switches (sw[7:6]) to produce a non-zero input, the program will move to either a line_draw, rect_draw, or circle_draw state, allowing the user to use their fingers to define two points for the corresponding shape. If a user presses btn[2] while holding their fingers, the program will fix the pixels corresponding to those points and store the resulting shape. If the user also has sw[2], the shape will be classified as movable. Otherwise, it will be considered a fixed shape. Until btn[1] is pressed, the program will remain in one of the draw states, but once the shape has successfully been stored, the program will return to the idle state. The user can repeat this process however many times they desire, but when they wish to start the simulation, they must press btn[1]. Once btn[1] is pressed, the physics engine is executed, the ball drops from its resting point at the top of the screen and interacts with the fixed and moving objects. Once the physics engine finishes executing (whether that results in the ball being on another object or off the screen entirely), the program returns to an idle state. At any point, while in the idle state, the user can press btn[2] to reset ("restart") the positions of all objects to their initial state (while still being on the screen–note this is different from "reset", which returns the user to a blank screen entirely with no non-ball objects present). In addition, users can use btn[0] to "reset" the sandbox.

II. Physics Engine (Younghun)

A. Representation Schemes

All numerical values within physics engine are handled in either 16 bits or 32 bits, with signed fixed decimal point representation. For convenience, we denote 16-bit representation as **SF16** or **SFIX16**, and 32-bit representation as **SF32** or **SFIX32**. As shown in diagram below, SF16 will have 1 sign bit, 10 integer bits, and 5 decimal bits. (i.e., decimal value x will be represented as an 16-bit signed integer, which has value of the nearest rounded value to $32x$). SF32 will have 1 sign bit, 20 integer bits, and 11 decimal bits.

SF16 and SF32 will support addition/subtraction, division, multiplication between same types, and the unary sqrt function. Addition, subtraction are straightforward integer arithmetic, with same overflow and underflow behavior. Division and multiplication should be handled with care of shifting decimal bits. We will use the same divider from the class, which will take 2 cycles, and use native xilinx multiplcation. Multiplication should be supported between SF16 and SF32, along with the same types, and it may output SF16 or SF32 depending on the circumstances. sqrt function is explained in the following section.



Most physical values, including position, velocity, sizes and masses, will be represented as SF16 in the memory, and may be converted to intermediate SF32 type during the computation. Time, which is used within the physics engine, is represented as SF32 for precision.

In-memory object representation scheme is described in section 4.a. Within physics engine, each object is decomposed to several primitive parts, to compute the collision. Each primitive part may be either circle and line segment, where point is described as a circle with radius 0. One object can have at most 8 parts - rectangles will have 4 circles and 4 lines, lines will have 2 circles and 1 line, and circle will have only one circle. Each primitive part may have at most 4 SF16

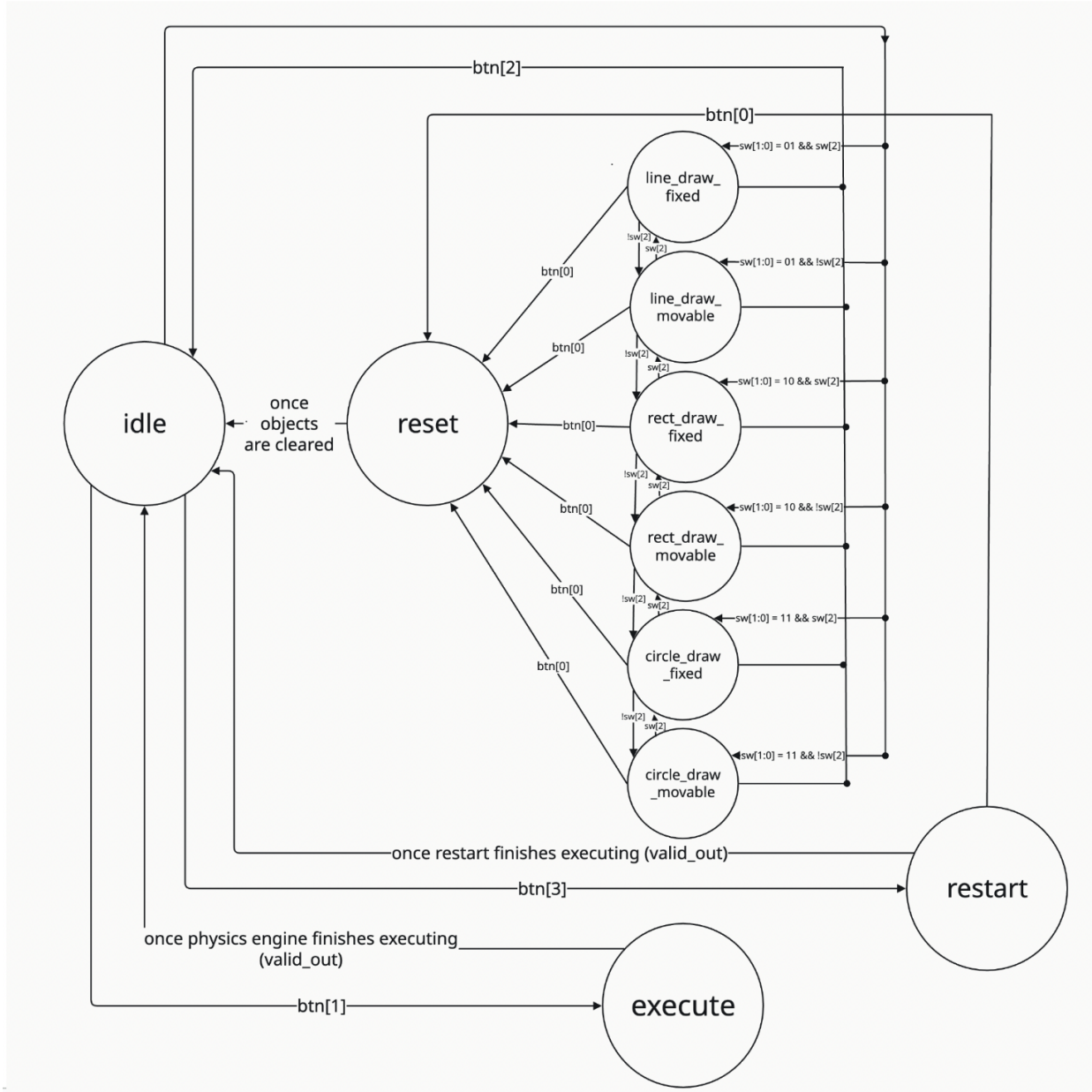


Fig. 1. Sandbox logic FSM

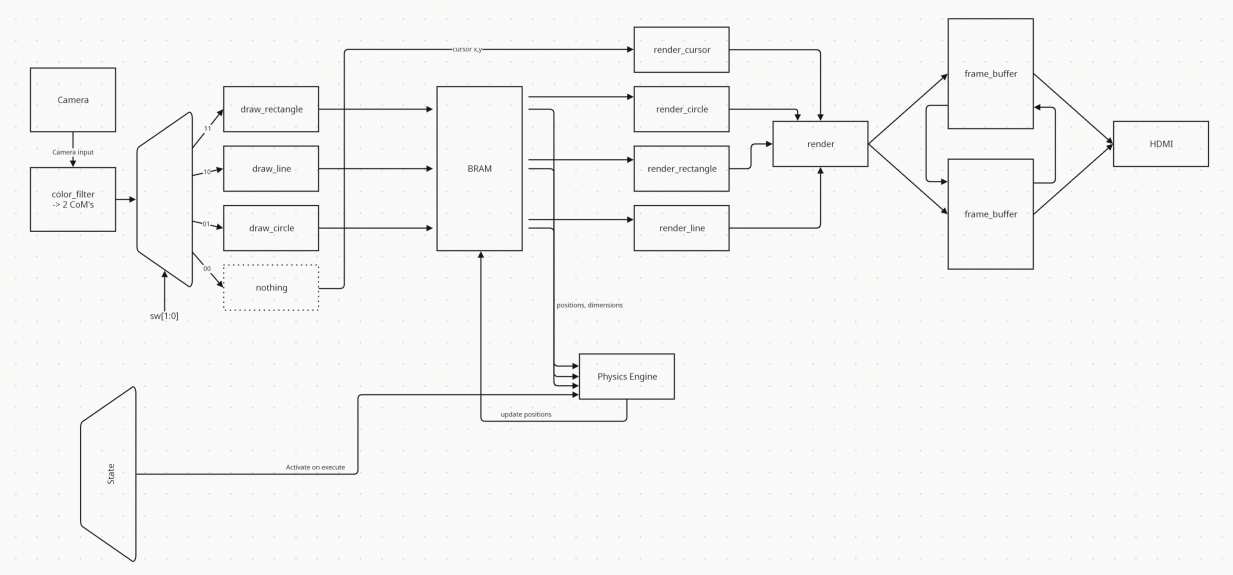


Fig. 2. Overall Block Diagram

values to specify, and it needs to have type indicator for empty, line, and circle. Therefore, we need 66 bits to describe each primitive part.

B. sqrt module (Daniel)

The sqrt module takes in a 32-bit signed fixed point decimal and computes the square root of it. There is a parameter, FRACTIONAL_BITS, which specifies the number of decimal/fractional bits in the input. The format of the input is therefore {1 signed bit, 31 - FRACTIONAL_BITS bits, FRACTIONAL_BITS bits}. With this information, a binary search algorithm is used to find the square root. There are 3 states in this module when it is trying to find/calculate the square root:

- mid_calc: this is where the middle value between the current low and high values of the iteration is calculated
- square_calc: this is where the middle value is squared, which is the value that is used to check against the input to see if the correct square root was found
- check: this is where the check is done to see if the square root was found, and then the output is set accordingly

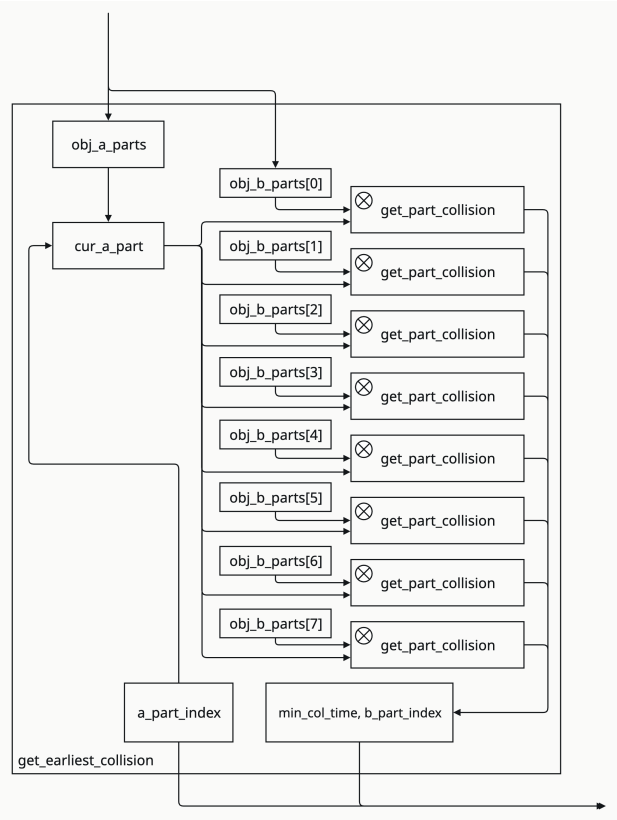
C. Atomic Computation Modules (Younghun)

Physics engine has three atomic computation modules, which will handle the math of each step between objects. For convenience, we denote the maximum number of objects as $OBJ = 16$, maximum number of parts per object as $PARTS = 8$, and maximum number of collisions in one frame as $MAXCOL = 64$.

Most costly module is `get_part_collision`, which does collision time calculation between two primitive parts. This module uses sqrt function, and should end in 50 cycles. This will run $OBJ * (OBJ - 1)/2 * PARTS * PARTS * MAXCOL$ times per frame, which is $12 * 23 * 8 * 8 * 64 = 1130496$. We will parallelize this module by factor of 8 for each object pair (so that each object pair will take at most 50*8 cycles), and by factor of 6 for each collision (so that we compute 6 pairs at one time).

Another module is `update_collision_vel`, which updates velocity for collision response. This should end in 16 cycles, since it has multiple divisions. This happens only once per collision, therefore this will run at most $MAXCOL = 64$ times per frame.

We also need a `update_collision_vel` module to update velocities and positions of each objects, given the time amount to proceed. Since this is simple multiplications, additions, and BRAM IOs, this should take 2 cycles per object. This will run at most $OBJ * MAXCOL = 1024$ times per frame.



D. Physical Details

Due to the unexpected integration complexities, we implemented the most fundamental free-fall logic, without collision. This simulates the simple equation of motion, $s(t + dt) = s(t) + vdt$.

The anticipated collision logic is more complicated. It decomposes the given object – circle, line segment or rectangle – into primitive parts – circle, or line segment – to calculate the collision time. We calculate the circle-circle collision time and circle-line collision time for all pairs of the parts, and use the earliest of them. Circle is decomposed into one part, line to three parts (a line and two points), and rectangle to 8 parts (4 lines, 4 points).

Circle-circle collision calculation is straightforward - we calculate part of the velocity vector parallel to the displacement of centers, and calculate the time until the centers get to the distance of the sum of their radii.

Circle-line collision calculation is done by calculated the inner product of the velocity and the displacement, based on the equation of the distance between the line and the point. We check the signed distance, and when if the center point ever enters the rectangle encompassing the line segment, padded by the radius of the circle. Note that we have to calculate circle-circle collision, because the endpoint of the line does not have the same collision response with the line itself. (i.e. circle hitting the edge of the line)

Collision response is calculated based on the mass and the angle of the impact surface. We can easily calculate the tangential and normal vector of the impact surface of the

circle or the line, and we reverse the relative velocity in that direction. We also scale down the resulting velocity based on the coefficient of restitution.

All above logics are implemented and tested in python, but only the first one is implemented in verilog. We have not implemented the torque, rotation or the normal/friction forces in python.

III. User Interaction (Daniel)

In our game, we allow the user the option to place objects on the screen before the game starts. These objects will be used to help get their ball to the target location (star, bucket, etc.) in order to win the game. In order to achieve this action, we will use color tracking to recognize two of the user's fingers, which will be used to move the object around the screen. There are 3 options of objects to place: rectangles, lines, and circles. sw[7:6] indicate which object is getting drawn:

- 00: nothing
- 01: circle
- 10: line
- 11: rectangle

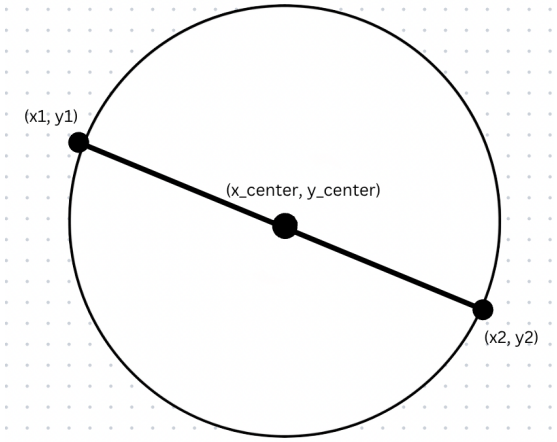
A. center_of_mass module

We use the previously written center_of_mass module (from lab 5) in order to track the user's fingers. The user will have two gloves on, with the tip of each index finger colored differently. One index finger will be colored pink, and the other blue. Therefore, we initialize two center_of_mass modules in the top_level, that will each use a different color channel for the mask: one module will use a chroma red channel and the other will use a chroma blue channel. As the user moves their fingers, the center_of_mass information will be getting sent to draw modules as inputs:

- x_com_1
- y_com_1
- x_com_2
- y_com_2

B. draw_circle module

The draw_circle takes in the 2 center_of_mass points, which are the two endpoints of a diameter on the circle. With this information, x1 and y1 are set to the coordinates of the center_of_mass point on the left, while x2 and y2 are set to the coordinates of the center_of_mass point on the right. Then, x_center and y_center are set to the center of the circle. From there, the differences of hcount from x_center and vcount from y_center are used to determine whether the pixel should be colored. The criteria for whether a pixel should be colored is as follows: $(\text{difference(hcount, x\_center)})^2 + (\text{difference(vcount, y\_center)})^2 \leq \text{radius}^2$. When btn[2] is pressed, the module then outputs the two endpoints of the diameter of the circle to get sent to storage ({x1, y1}, {x2, y2}).

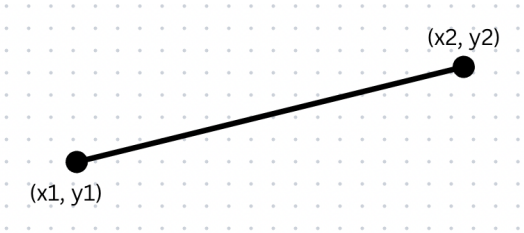


C. draw_line module

The draw_line module takes in the 2 center_of_mass points, which are the two endpoints of the line. With this information, x1 and y1 are set to the coordinates of the center_of_mass point on the left, while x2 and y2 are set to the coordinates of the center_of_mass point on the right. From there, we check the slope of two lines:

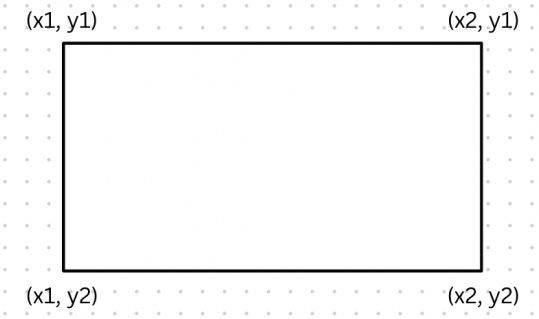
- One line with endpoints (x1, y1) and (hcount, vcount)
- One line with endpoints (x1, y1) and (x2, y2)

If these two slopes are the same within a margin of error, then the pixel at (hcount, vcount) should be colored in. When btn[2] is pressed, the module then outputs the two endpoints of the line to get sent to storage ({x1, y1}, {x2, y2}). Example below:



D. draw_rectangle module

The draw_rectangle module takes in the two center_of_mass points, which are opposite corners of the rectangle. With this information, x1 is set to the smaller x value and x2 is set to the larger x value. Same thing is done with the respective y values. From there, the pixels that get colored in are the ones where $x1 \leq hcount \leq x2$ and $y1 \leq vcount \leq y2$. When btn[2] is pressed, the module then outputs the four corners of the rectangle to get sent to storage ({x1, y1}, {x2, y1}, {x1, y2}, {x2, y2}). Example below:



IV. Object Storage (Disha)

The object_storage module interfaces with all of the other key modules in our project (e.g. user interaction, rendering, physics engine) and is comprised of several sub-modules to encompass all of its functionality. object_storage critically houses the four identical BRAMs that store the data of all of the objects on the screen. Thus, once a user creates an object to their liking, the draw modules send object_props, a 87-bit array structured as follows:

- $[86 : 86]$ is_static (to indicate whether or not the object is affected by physics during the simulation).
- $[85 : 84]$ id_bits (used to identify the type of the object– this is the output of sw[7:6] in section III.
- $[83 : 0]$ contains the four 21-bit numbers representing the x and y coordinates of the up-to-four critical points of the object (i.e. in the case of a rectangle, this splice would contain a concatenated bit representations of all four vertices in $x_1y_1x_2y_2x_3y_3x_4y_4$ fashion). If not all four 21-bit numbers are needed, the critical points will be placed in the most significant bits of the list, followed by trailing zeroes. For instance, in the cases of a line and a circle, only two 21-bit numbers are needed (the endpoints of the line and a diameter of a circle, respectively).

Since the object_storage module is used for reading from and writing to the BRAMs, we use two flags, is_write_valid and is_read_valid, to determine which of the two actions (if any) are being requested. In the case that a module is requesting to read information, object_storage also takes a packed [3:0][6:0] current_addr, which can accommodate up to four read requests at once.

Regardless of whether or not the object_storage module is accepting a read or a write request, it outputs the parsed information from object_props as it is being stored in memory to be used or discarded by the other modules.

Internally, the object_storage module is comprised of one instance of draw_to_storage_conversion module and four instances of the storage_breakdown module, as well as four instances of xilinx_true_dual_port_read_first_2_clock_ram in order to create four dual-port BRAMs. (We decided on dual-port over single-port, since the render and physics engine modules will both frequently need to read from and write to each BRAMs, likely at the same time at certain points.) The storage_module just de-concatenates the final value that is stored in the BRAM into a format that is usable for the render and physics engine modules. The draw_to_storage_conversion module, on the other hand, requires additional clarification.

A. draw_to_storage_conversion

This module handles converting object_props into the format that they are stored in the BRAMs and returns the de-concatenated elements, as they are returned by the object_storage module. Specifically, the 103-bit width of each BRAM entry is distributed as follows:

- $[102 : 102]$ is_static (same as before)
- $[101 : 100]$ id_type (also same as before)
- $[99 : 84]$ pos_x (described below)
- $[83 : 68]$ pos_y (described below)
- $[67 : 32]$ params (described below)
- $[31 : 16]$ vel_x (velocity in the x-direction, as discussed in section II. This module initializes it to 0).
- $[15 : 0]$ vel_y (velocity in the y-direction, as discussed in section II. This module initializes it to 0).

Since each shape has at least one critical point (a line has two endpoints, a circle has a center–which we calculate by taking the average of the two diameter endpoints–and a rectangle has four vertices), we decided to store it (with pos_x and pos_y both stored as 16-bit numbers to standardize with the physics calculations) separately as a way of anchoring the object and stored the rest of the data under params in order to standardize the process across different shape types. As such, params varies across object types:

- Line: this is the simplest case; params simply contains the concatenated x_2y_2 (i.e. the other point in the line).
- Circle: params contains the value of the radius, which we calculate by taking the sum of the squares of the differences between the center of the circle and one of the measured critical points, square rooted (i.e. $\sqrt{(pos\_x - x_1)^2 + (pos\_y - y_1)^2}$. To calculate the square root of this quantity, we used the sqrt module defined in section II(b).
- Rectangle: this was the most complex case because of the manner in which we decided to store the remaining critical points. A rectangle requires at least five quantities to be identifiable (e.g. two x-coordinates, two y-coordinates, and an angle). However, to reduce the amount of space required and store the data in a way that is conducive for the physics engine and for the future possible inclusion of rotation (currently, all of our rectangles are parallel to the screen, but we hope to introduce rectangles of all angles), we decided to store the top-left point (pos_x and pos_y), then find the next two points clockwise from the first (first_clockwise_x, first_clockwise_y) and (second_clockwise_x, second_clockwise_y), and store the displacements (dx_1 and dy_1 in the x and y-directions between the first clockwise point and pos, as well as dy_2, the displacement between the first and second clockwise points. To help with this, we made an additional helper module, nth_smallest, which uses takes four points and returns them sorted using a bitonic sort. Then, the first

element is pos, while the first element with a larger y than pos is first_clockwise.

V. Rendering and User Experience (Disha)

To account for the difference in clock speeds between the rest of the pipeline (100 MHz) and the HDMI (74.25 MHz), we decided to use two rotating 360p frame buffers, one that stores the current data for the frame, and one that stores the data from the last frame. The HDMI would be fed the data from the previous screen while the current screen would be written. We would additionally be erasing pixels immediately after they are read to keep the frames consistent. This data would then be scaled up four times to fit the standard 720p display screen. To prioritize user experience, We decided to use a black background (i.e. value of 0) and two colors to represent all of the objects on the screen (blue if static, red if dynamic) in order to minimize the amount of additional storage required for rendering. However, this doubled the number of bits we needed to store the pixel data (1 for black and white versus 2 for three colors), and hence, we ran out of FPGA storage on a full resolution frame buffer and decided to stick to 360p in the frame buffer and scale up on the screen.

The render framework is largely handled in the render module, which takes in start and end addresses (which mark where the objects are stored in the BRAM), and then instantiates the object_storage module to read four simultaneous objects from the four BRAMs. Once the objects are recovered, the information is passed into separate render_object modules based on the id_type of the object. The render_object module sequentially returns a new pixel that is contained in the object every clock cycle. These four points are then concatenated by the render module and read at the top level, at which point they are fed into the frame buffer one pixel at a time. Even though the process is bottle-necked by the frame buffer's capacity to only process one pixel at a time, determining whether or not a pixel is in an object takes longer than one clock cycle on average (because rectangles require four comparisons, one for each distinct line that makes up the set of edges and circles require at least two to account for pipelining the $(x - pos\_x)^2 + (y - pos\_y)^2$ quantity that is checked against the radius), and hence having four BRAMs instead of just one is advantageous.

Acknowledgment

We would like to thank Jan Park and Professor Joe Steinmeyer, for their continued guidance throughout this project, as well as all of the lab assistants and teaching assistants for 6.205 (in particular, but not limited to, Hasan Zeki Yildiz, for their thoughtful insight into the render design process and Kiran Vuksanaj for their continued guidance in understanding the specifics of Verilog). We would additionally like to thank Pleng Chomphoochan for the same.