FPGAbility: Spatially-Aware Writing

Annie Liu
Department of Electrical Engineering and Computer Science
Cambridge, MA
annieliu@mit.edu

Kristoff Misquitta
Department of Aeronautics and Astronautics
Cambridge, MA
kmisqui@mit.edu

***Abstract*—We propose FPGAbility, a 3D rendition of Notability. The user shines a laser at a camera mounted to servos and wired to the Nexys 4 DDR FPGA. This light serves as a stylus that will record strokes at its location, which will be displayed on a monitor. The speed of the laser determines the size of the strokes on the monitor. This is in addition to features like writing with multiple lasers simultaneously, erasing, and simple creation of geometric primitives like lines, circles, and curves. Further, when the camera moves out of the frame from where the original writing was, the writing will move out of frame as well. When the camera is back in the frame of the original writing, the writing will re-appear, giving the illusion of panning in space.**

***Index Terms*—Field programmable gate array, video processing, digital systems**

I. Problem Statement (Kristoff)

Notability is among the highest-rated digital note-taking software for iPad. Features like drawing, erasing, construction, and infinite scrolling creates a writing experience that replicates – and often enhances – that offered by pencil and paper. Nonetheless, being restricted to a tablet screen, Notability is inherently two-dimensional, limiting its adoption in novel, three-dimensional contexts like augmented or virtual reality.

As a proof-of-concept of the ability to write in 3D space, we integrated a set of drawing and graphics transformation modules onto an existing video processing architecture on the Nexys 4 DDR FPGA. As opposed to pencil-on-paper (or stylus-on-tablet), we use a laser-on-canvas method for detecting strokes, which are then stored and transformed freely as the camera pivots with two degrees of freedom on a servo-controlled mount. In addition to basic writing and erasing functionality, we aimed to enhance Notability's core capabilities with additions such as rapid creation of geometric primitives (lines, circles, curbes) and making stroke widths dynamic with laser speed for organic script. As an additional challenge, we designed our architecture to be as scalable as possible, currently supporting the simultaneous use of two lasers for writing (red and green) as well as a third for erasing (blue) in any mode.

II. System Architecture

A. Overview (Kristoff)

The key storage element in our design is the field buffer: a shallow but large BRAM spanning the camera's entire field of view and storing all detected laser strokes of a given color. It operates with the existing frame buffer, a small but deep BRAM spanning only the current field of view and storing all

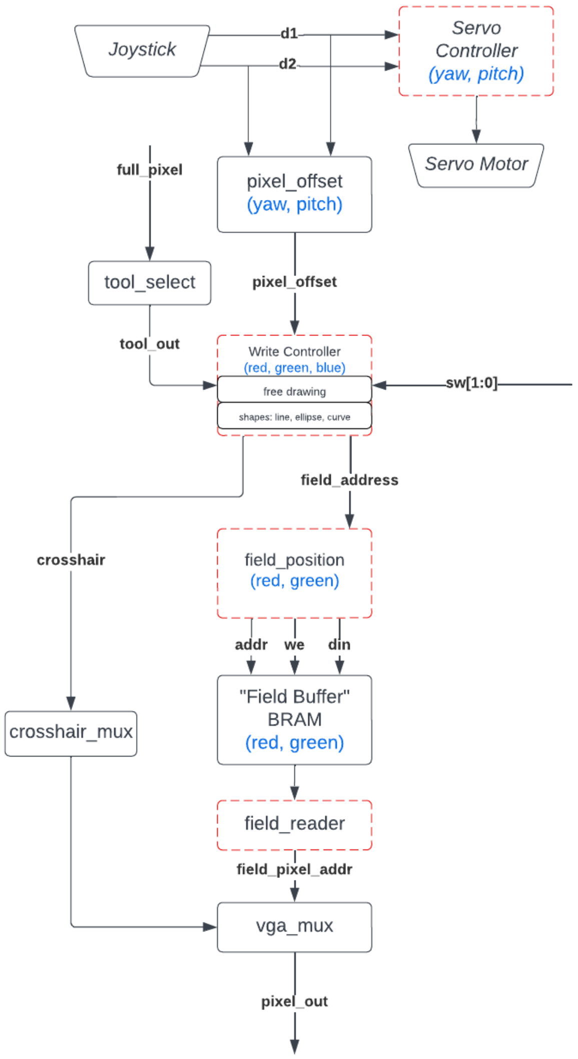

Fig. 1. High-level diagram of FPGAbility. Multiple instantiations of modules annotated in blue but not drawn. Modules outlined in red are examined in depth later in the report.

the color information the camera receives. Note the differing nature of these memory elements: the field buffer serves as long-term storage, maintaining a record of strokes until reset or erasure, while the frame buffer is continuously refreshed with new camera frames.

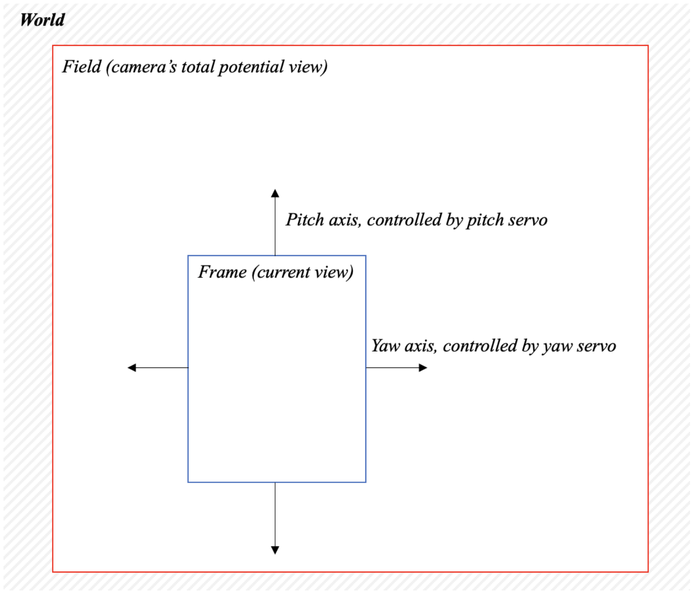
World
Field (camera's total potential view)
Pitch axis, controlled by pitch servo
Frame (current view)
Yaw axis, controlled by yaw servo

Fig. 2. Arrangement of field, the total potential view of the camera, and frame, the camera's current view. Yaw and pitch servos are used to navigate the static field.

As the servos rotate, changing the camera's orientation, the addresses accessed in the field buffer change accordingly (Fig. 2). This enables extended writing limited only by the memory cap on the FPGA.

The four core functionalities of our design – namely servo control, free drawing, geometric primitive creation, and field positioning – all compete to interface with the field buffer to produce strokes and shapes that freely transform in space. We imposed hierarchy among these features by extensive use of the major-minor FSM paradigm. Specifically, top-level modules like `write_controller` and `shapes` delegate work to a suite of continuously running submodules. Point coordinates and valid output signals are then multiplexed before conveyance to the field buffer for long-term storage.

B. A Note on Simultaneity (Kristoff)

In keeping with the spirit of FPGA design, the strength of our architecture derives from its ability to perform multiple tasks in parallel. For example, `servo_controller` is instantiated twice, as both a yaw controller and a pitch controller manipulating two axes independently (Fig. 3). `write_controller`, likewise, is instantiated as a `red_write_controller`, `green_ write_ controller`, and `blue_ write_ controller` supporting, at once, blue erasing and any writing mode for the red and green lasers (Fig. 3). In code, the field buffer, too, is comprised of two BRAMs, one for storing red strokes and the other for storing green.

Examining data flow through one of these many parallel branches is sufficient for understand FPGAbility's architecture, so this report omits duplications in its block diagrams.

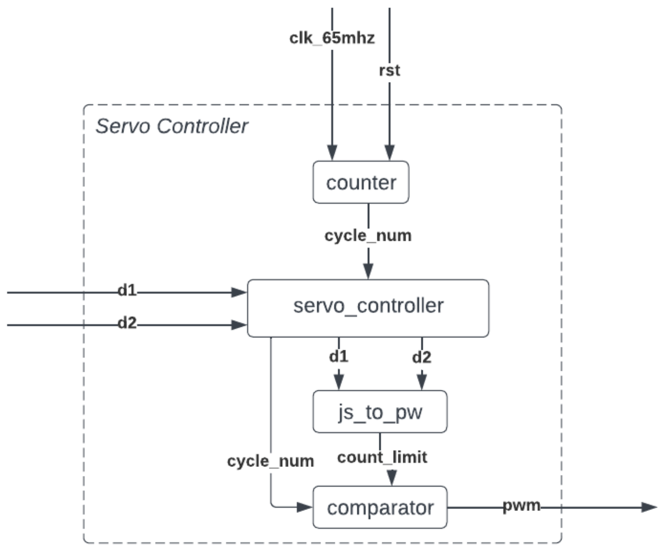

Fig. 3. Closer view of `servo_controller`. Note that this architecture is instantiated twice: once for the yaw servo and once for the pitch servo.

C. Servo Controller (Kristoff)

The two SG90 servos are oriented to provide freedom in the yaw and pitch axes. Aside from their individual joystick mappings (yaw servo responds to right/left joystick inputs, pitch servo responds to up/down inputs), the control of each is identical. Specifically, each accepts a 50 Hz input signal (20 ms period), and the first 1-2 ms can be modulated to encode the desired angle. The required pulse-width modulated signals are supplied by parameterized `yaw_servo_controller` and `pitch_servo_controller`. The modules described below are submodules of these two.

`Counter` establishes the base 50 Hz repetition by repeatedly counting $\sim$1.3 million positive clock edges, and then resetting to 0. This method (as opposed to techniques like Clock Wizard) was selected due to the extremely low frequencies involved and its sufficient accuracy for servo control. In parallel, based on the duration for which any given direction of the joystick is held, `js_to_pw` sets the number of counts for which the signal routed to the servo should remain high, ranging from about 65,000 (1 ms high, or turn to 0º) to 130,000 (2 ms high, or turn to 180º). The output of `counter` and the `js_to_pw` count limit is compared within `comparator`, which outputs high to the servo so long as the current count is below the threshold.

D. Field Positioning (Kristoff)

Not only must the camera detect and store strokes in a given frame, but it must also translate strokes and "remember" their original positions when the camera is panned. This concept was implemented as an additional shallow but large (2 bits, 900x974) "field buffer" BRAM, distinct from the existing deep but small (16 bits, 320x240) "frame buffer."

The location in the field buffer to which a pixel is written depends on the position of the servos. Concretely, with each servo initialized to 90º on reset, the center of the field buffer is modified and read. However, rotating both to 0º ("looking up and left") superimposes the frame on the top left corner of the field buffer.

The bridge linking servo positions to field offsets is `pixel_ offset`, which uses the same joystick inputs and a number of mathematically derived relations (pixels per degree, servo speed) to convert a joystick hold duration to the number of pixels by which to translate. Again, this occurs simultaneously for the yaw and pitch servos, the former dictating `yaw_offset` and the latter dictating `pitch_offset`, prior to flattening for interfacing with the BRAM.

`field_position` writes the output of the masked camera field of view into the field buffer as described above (Fig. 1). For a simple canvas, like a plain wall or whiteboard, this will exclusively be the shine of the laser.

As `field_position` writes into the field buffer, `field_ reader` retrieves information from the same dual-port BRAM for delivery to `vga_mux`. The same linear transformation from hcount and vcount to a linear index using the servo offsets is applied.

E. Write Controller

1) Unified Architecture (Kristoff): Writing may occur in any of four modes: free drawing, line generation, ellipse generation, and curve generation. The modules for each mode are carefully designed to fit – with the same inputs and outputs – into a top-level module called `write_controller` (Fig. 4). As this pipeline is shared across all forms of writing, it is illustrative to examine the flow of a pixel through it now.

`tool_select`, the old `threshold`, uses red chrominance, blue chrominance, and green masks to identify, with as little overlap as possible, the signatures of red, blue, and green lasers on a per-pixel basis. It feeds the classification of each pixel in synchrony with hcount and vcount into `center_of_mass`. Depending on whether it is within the red, green, or blue `write_controller`, `center_of_mass` will compute the center of the corresponding points. This center of mass is then fed into one of the four writing modes to form the basis of each of their strokes. The output of each of the writing modes is a sequential stream of points to be addressed into the field buffer. Which of the four modes is sending its points to the `write_controller` output is determined by the states of two switches on the FPGA.

2) Free Drawing and Dynamic Stroke Widths (Kristoff): In the original conception of free drawing, points outputted by `center_of_mass` were written immediately to the field buffer, resulting in thin, illegible strokes and numerous errant pixels (from the environment and not the laser's shine) in the frame. To resolve the latter issue, `center_of_mass` was modified to additionally detect whether the laser is on via checking whether the number of masked pixels is greater than a threshold. A high number of pixels indicates the presence of a concentrated, bright object; in other words, the laser.

To resolve the visibility issue, both `width_select` and `thicken_point` were added. `width_select` reads in two centers of mass and, based on the Manhattan distance between them, outputs a number representing the extent to which the second one should be padded when displayed. Larger distances correlate roughly with faster laser motion and, as in real life, fatter strokes. This number is converted to actual padding using `thicken_point` , which assigns rectangular bounds on hcount and vcount within which points should be drawn as the frame is scanned. Behavior at the edges is handled by casework to avoid overflow.

3) Geometric Primitives (Annie): A set of geometric primitives is a useful starting point for designs and text; for example, circles for Venn diagrams or lines to simulate ruled paper. Thus, in addition to free-drawing, FPGAbility implements a mode in which users can draw straight lines, curves, circles, depending on the values of switches 0 and 1 (for red) or switches 2 and 3 (for green).

Lines were implemented using Bresenham's straight line algorithm. The algorithm requires the two end points of a line as inputs. The design decision we made was to select the first point $(p1x, p1y)$ as the position where the user first shone their laser and the second point $(p2x, p2y)$ as the position where the user last shone their laser.

The `line_gen` module is what selects the two points. These two points are then fed into the `plot_lines` module, which implements Bresenham's line algorithm. When two points are inputted into the module and `valid_pixel_in` is high for one clock cycle, the necessary variables are initialized. The algorithm starts at point $(p1x, p2y)$ and attempts to reach $(p2x, p2y)$ through a series of error calculations. In every clock cycle thereafter, until the current (x, y) has overshot $(p2x, p2y)$, `valid_pixel_out` is high, and *px* and *py* are set to the x and y values of the current point. Depending on the color of the laser, the outputs of `plot_lines`—*px*, and *py*—are fed into either the red or green BRAM, same as described in the previous section.

Similar to the `line_gen` module, the `circle_gen` module is what selects the two points. The center of the circle (xm, ym) to be drawn is the first point $(p1x, p1y)$ selected, whereas the radius is the Manhattan distance from the first point $(p1x, p1y)$ to the second point $(p2x, p2y)$. Both the coordinates of the center of the circle and the radius are fed into the `plot_circle` module, which implements a standard circle generation algorithm. Similar to the format of the `plot_lines` module, when two points are inputted into the module and `valid_pixel_in` is high for one clock cycle, the necessary variables are initialized. In every clock cycle thereafter, until the algorithm completes, `valid_pixel_out` is high, and *px* and *py* are set to the x and y values of a point that lies on the circle. The outputs of `plot_lines`—*px*, and *py*—are fed into either the red

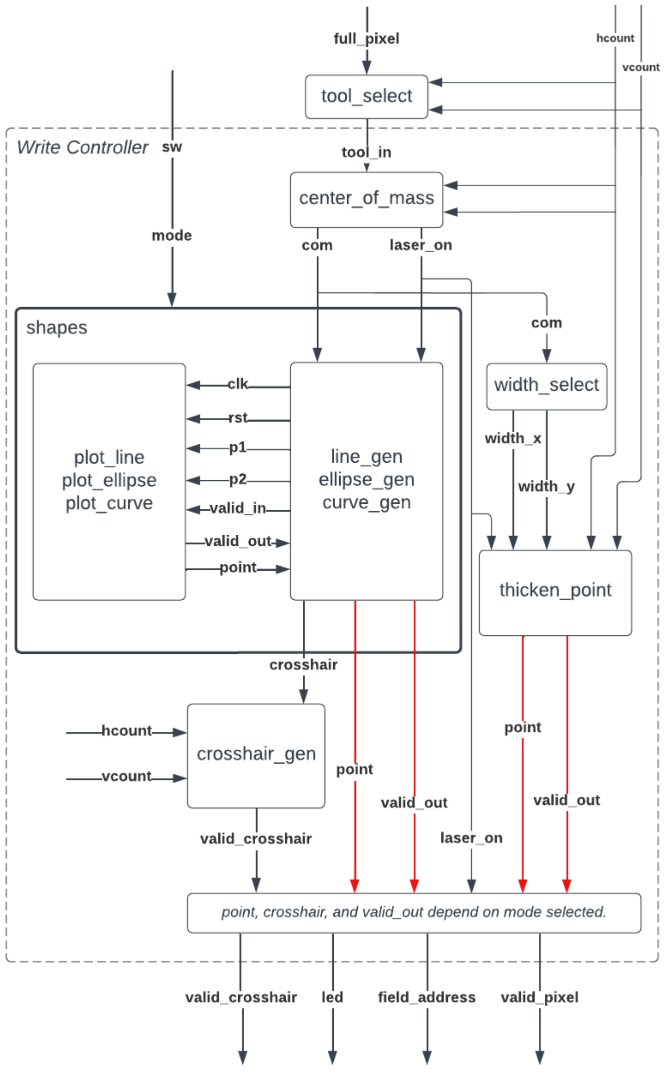

Fig. 4. Deeper view of `write_controller`. Note that `line_gen`, `circle_gen`, `curve_gen`, `plot_line`, `plot_circle`, and `plot_curve` are all distinct modules, but they have identical inputs and outputs so are compressed to the same blocks for readability.

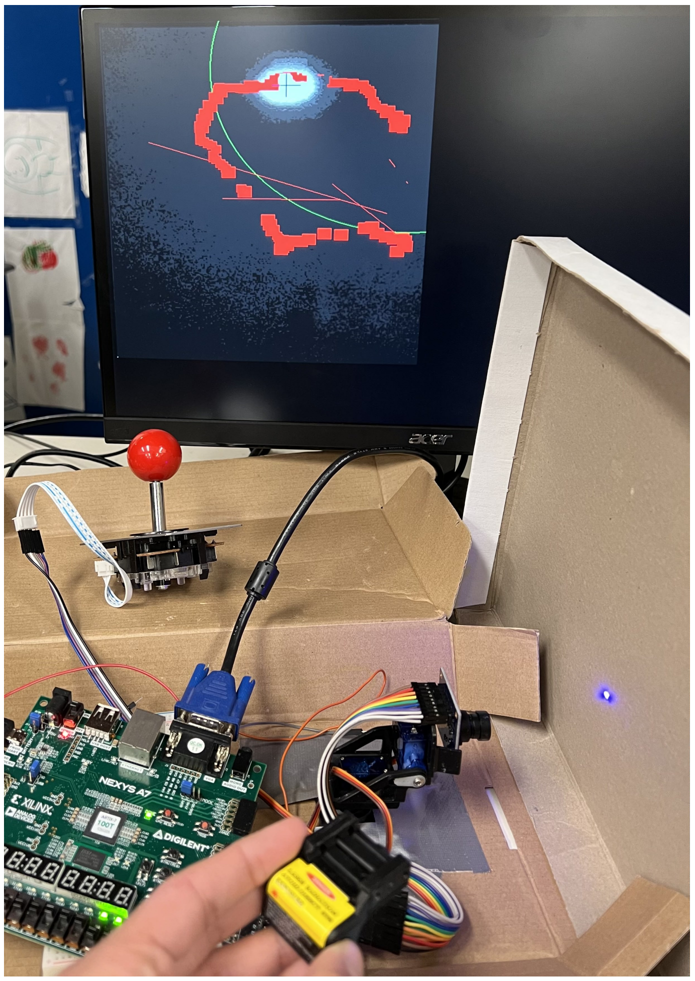

Fig. 5. The setup used for writing. Note the two LEDs lit up at the bottom corner to indicate detection of a blue laser, as well as the blue crosshair erasing the red stroke in the frame. The outputs from the green circle generator, red line generator, and red free draw are shown.

or green BRAM, depending on the color of the laser, same as described in the previous section. Our architecture is structured in such a way that the circle is still drawn even if the points do not fit in the current frame. The out-of-frame parts of the circle can be seen by using the joystick to move the circle into frame.

Like both the `line_gen` and `circle_gen` modules, the `curve_gen` module selects two points, $(p1x, p1y)$ and $(p3x, p3y)$, that serve as the end points of the curve that is to be drawn. Bezier's curve algorithm, implemented in `curve_gen`, however, accepts three points. The third (middle) point $(p2x, p2y)$ is set as $p2x = p1x$ and $p2y = p3y$. These three control points form a right-angle corner: the line formed by connecting $(p1x, p1y)$ and $(p2x, p2y)$ and the line formed by connecting $(p2x, p2y)$ and $(p3x, p3y)$ are perpendicular to each other and tangent to the curve drawn using Bezier's algorithm.

The `shapes` module selects between these geometric shapes depending on the switch value, simplifying the code in `top_level` by tying them together. An instance of `shapes` itself is placed in `write_controller`; depending on the switch values, the user can either free draw or create geometric shapes.

III. Calculations (Kristoff)

Extensive calculations were performed to estimate memory usage and optimize servo control. The ones most critical to the design are detailed here.

A. Field Buffer Size

Yaw freedom: 38° in each direction
Pitch freedom: 30° in each direction
Camera frame size: 240px $\times$ 320px

Pixels per degree [1]: 9.5 px/deg

BRAM x size: $240\text{px} + 2 \times (9.5\frac{\text{px}}{\text{deg}} \times 38°) = \boxed{962\text{px}}$
BRAM y size: $320\text{px} + 2 \times (9.5\frac{\text{px}}{\text{deg}} \times 30°) = \boxed{890\text{px}}$
BRAM Total Size: $2 \times 962 \times 890 = \boxed{1,712,360 \text{ bits}}$

Frame Buffer Size: $16 \times 240 \times 320 = 1,228,800$ bits
FPGA BRAM Capacity: $4,860,000$ bits $> 2,941,160$ bits

B. Servo Limits

Time per positive edge: $\frac{1}{65\times10^6} = 15.38\text{ns}$

Counts per servo period: $\frac{20\text{ms}}{15.38\text{ns}} = 1,300,390$ counts

Counts for 0: $\frac{1\text{ms}}{15.38\text{ns}} = \boxed{65,020 \text{ counts}}$

Counts for 180º: $\frac{2\text{ms}}{15.38\text{ns}} = \boxed{130,039 \text{ counts}}$

IV. Validation (Annie)

A. BRAM, DSPs, and LUTs

According to the reports produced by Verilog, 102 out of 135 BRAMs were used for a total utilization of 75.56%, and 59 out of 240 DSPs were used for a total utilization of 24.58%. A total of 6299 LUTs are used for a total of 9.94% utilization, with 9.93% as logic and 0.03% as memory.

B. Timing Constraints and Critical Path

For many stages in design, our critical path did not meet timing constraints. The critical path began at the `pitch_pixel_offset` module, which produces the correct pitch offset given input from the joystick, continuing onto the `tool_select` module, which selects which controller to use based on which color the camera picked up. This is fed into the `red_controller` module, then `crosshair_m`, which produces a red crosshair wherever the user is pointing the red laser. It then makes its way to `red_field`, which marks the end of the critical path. To meet timing constraints, we added a register after the `tool_select` module and another after the `red_controller` module, bringing our worst negative slack to 0.274 ns and a total negative slack of 0.00 ns.

C. Servo Validation

The FPGA's digital output was connected to an oscilloscope to verify that the servo specifications were implemented accurately. The signal demonstrated the expected 1-2ms modulation without exceeding limits.

D. Geometric Curves Validation

To validate the geometric primitive modules, we translated the algorithms into Python and wrote test benches for `line_gen`, `circle_gen`, and `curve_gen`. The output of the Python program and the test benches match for the same inputs, serving as a useful sanity check in a familiar language.

[1]Calculated by marking the limits of the camera view on a paper at a given distance away, then using simple trigonometry and the known frame size.

E. Scaling in the Real World

Sharply increasing clock frequency would shorten timing allotments for our calculations. Given the small margins in our worst negative slack, this would very likely require optimizing the calculations we perform as VGA scans over the image, or where those calculations are performed (focusing more on the end of the frame or surrounding blank space). One area for focus here is the geometric primitives modules: (`line_gen`, `circle_gen`, `curve_gen`), which involve long chains of logic squeezed into the video timing. There are opportunities to segment those calculations over more cycles or refreshing the screen with updates more slowly.

It's also worth noting that we are limited by the camera's maximum ∼25 MHz clock (currently it's running at 16.25 MHz).

There are many ways FPGAbility can scale. The reports produced by Verilog state that 75.56% of the total BRAM storage is utilized; with 24.44% additional space left, we could add another BRAM to incorporate another user, slightly sizing down the others in the process. This additional space could also be used to expand the size of the servo's field of range or the size of the output that appears on the screen. In the spirit of Notability, we could also have an option for the user to save their current drawings to be reopened at a later time. The number of drawings we can save is proportional to the amount of space we have.

All of these features–or a conglomeration of them–would be possible with more space. We could also make do with less space, but up to a limit, since we need at least 102 blocks to store the two field buffers and the one frame buffer. In the case that space is a limiting factor, we would remove one of the field buffers and allow for only one user to draw.

V. Reflections (Kristoff)

A. Goals

We laid out the following goals for FPGAbility:

- **Commitment**: integrate joystick, servo, and laser for writing that can move in and out of frame.
- **The Goal**: more complete writing experience. Erasing, stroke colors, and dynamic stroke widths.
- **Stretch Goal**: handle multiple simultaneous users (two lasers in field of view, each with different colors to capture different functionalities)

We implemented all the features delineated in our commitment, goal, and stretch goals, even achieving the integration of three lasers, rather than our initial target of two. Inspired by this success, we reached a few steps further to create an entirely new set of modes for our application: geometric primitives creation, including lines and circles. We were thrilled to make an application closely paralleling nearly every major feature of Notability on the FPGA fabric.

We designed our code around the idea of scalability, so it's foreseeable that on a more powerful FPGA, and with more careful memory optimization, an additional set of lasers could be integrated with little effort (instantiation of another write controller). The concept of multiple users writing in space, united by an underlying set of transformation modules, is a powerful one that holds promise is a meta-oriented world.

B. What to Improve

The OV7670 camera is highly sensitive to ambient light and shadow, skewing the center of mass calculations. It's recommended that the setup is operated in a dark environment or with potential sources of glare shielded.

Drawing curves seems to only work for small curves, but attempting to draw a larger curve produces no results. The test bench for the `plot_curves` module matches the output produced by the Python program for the inputs we tested. Given additional time, this is something we would look more closely into.

In some servo orientations, there is also a horizontal strip where shining a laser shows the expected crosshair on the screen, but the drawings do not appear. This problem seems to affect both the free draw and geometric shapes. We believe this might be due to an overflow of pixel values, the use of signed registers, or because the field buffer has trouble reading from or writing to specific addresses.

C. Contributions

Kristoff focused on control of the servo. He also implemented the base writing/erasing functionality, dynamic stroke widths, and simultaneous lasers. Annie integrated the joystick, tied servo and field buffer with field positioning, and developed the geometric primitives code, including lines, curves, and circles. Debugging modules and major architectural decisions were made jointly.

FPGAbility's repository is accessible at: https://github.mit.edu/annieliu/6.205-final.

Acknowledgments

We'd like to thank Prof. Joe Steinmeyer for his guidance on this project, as well as Fischer Moseley and Jay Lang for their mentorship and contributing the video processing code.