

Sketch-PGA Final Report

Emma Chen

Department of EECS

Massachusetts Institute of Technology

Cambridge, Massachusetts

emmachen@mit.edu

Maya Jyothinagaram

Department of EECS

Massachusetts Institute of Technology

Cambridge, Massachusetts

mayajyo@mit.edu

Eileen Zu

Department of EECS

Massachusetts Institute of Technology

Cambridge, Massachusetts

zuleen@mit.edu

Abstract—We present the design for a virtual transparent whiteboard system implemented on FPGA. Utilizing a camera module to produce a high-definition livestream on HDMI, users are able to draw onto the screen and toggle between different brush settings, colors, and modes using hand gestures. The hands are tracked using colored foams on finger tips, and the locations of each finger is determined via a K-means clustering algorithm on a real-time video feed.

Index Terms—Digital Systems, Field Programmable Gate Array, Gesture Tracking, K-Means

I. INTRODUCTION

Virtual whiteboards are often used to enhance interactions between the user and the observer. However, virtual whiteboards can cost upwards of thousands of dollars to purchase, can be easily damaged by improper use, and are large and not easily transportable. We were heavily inspired by the Disney Channel wand ID opening scenes and the transparent whiteboards used in virtual teaching formats. Thus, we wanted to create a virtual whiteboard on the FPGA that would remedy all of the issues that plague commercial virtual whiteboards and provide a useful tool for users.

Our project is split into 4 main components: the high-definition video pipeline, draw module, a dynamic K-means module, and an interface graphics module.

Using a pink-foam tipped stylus, the camera tracks the stylus using color thresholding and overlays pixels of a specified brush width and color at that location. Pixel information is stored in a canvas buffer BRAM, addressed using the center of mass x- and y-coordinates. The image currently sitting in this canvas buffer is rendered to the screen via the video pipeline, allowing the brush strokes of the user to be visualized in real time. We use a dynamic K-means algorithm to track the user's fingertips, enabling the user to set the color and size of the brush and switch between draw, erase, and stamp mode depending on the number of fingers they hold up. The graphics interface provides a visual display to the user, overlaid on the video feed. It indicates what color, brush size, and mode (draw, erase, stamp) is currently enabled by the user. Additionally, it also notifies the user of how many fingers it sees the user holding up via the k-means "battery." This provides further visual feedback to the user on what settings are active when interacting with our system.

III. DRAW MODULE (EILEEN)

II. HIGH-LEVEL DESCRIPTION

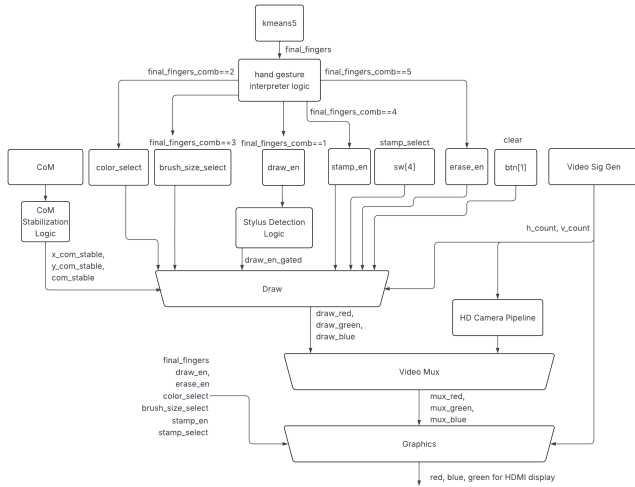


Fig. 1. High Level Block Diagram

TABLE I
SELECTABLE COLORS

Color	RGB Value	Encoding
Transparent	000000	4'b0000
Red	FF0000	4'b0001
Blue	0000FF	4'b0010
Green	00FF00	4'b0011
Orange	FF9800	4'b0100
Dark Violet	9400D3	4'b0101
Black	0A0A0A	4'b0110
Yellow	FEDB00	4'b0111
Turquoise	40E0D0	4'b1000
Dark Green	008000	4'b1001
Light Blue	87CEFA	4'b1010
Maroon	800000	4'b1011
Brown	8B4513	4'b1100
Blood Orange	FE3E00	4'b1101
Hot Pink	FF69B4	4'b1110
White	FFFFFF	4'b1111

A. Drawing, Erasing, and Clearing Modes

The drawing module has five main input signals related to the drawing: *draw_en*, *erase_en*, *clear*, *color_select*, *brush_*

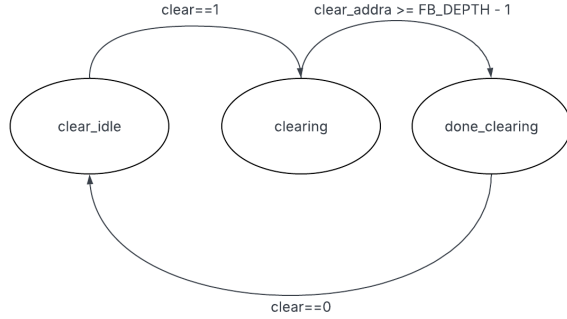


Fig. 2. Clear finite state machine

size_sel. The first three signals are used as triggers for various modes, such as drawing, erasing, or clearing the entire canvas. *color_select* is implemented by encoding 4-bit values to represent the 16 different colors as seen in Table 1. *brush_size_sel* is implemented using 2-bit values and to encode four different brush sizes: 3-pixels, 5-pixels, 7-pixels, or 9-pixels widths.

Alongside these, the draw module also takes in *x_com*, *y_com*, *h_count*, *v_count*, and a *com_stable* signal. The x- and y-coordinates for the center of mass are downsampled by a factor of four for memory efficiency, and these downsampled values are then used to calculate the BRAM address at which a drawn pixel is stored. The value stored at each address is the value of *color_select* which is used to index into the lookup table in the draw module for the appropriate color to output via the *pixel_red*, *pixel_green*, and *pixel_blue* channel. Enabling erasing is very similar to drawing – the only difference is that the *color_select* value written to the BRAM is set to be 4'b0000. This sets all output channels to 0, so that the *video_mux* will choose to show the camera feed rather than overlaying a drawn pixel.

Clearing is triggered when button[1] is pressed on the FPGA. This flushes the entire canvas of anything that was drawn. A finite state machine is used to handle the transitions between the idle and clearing states. It iterates through every address in the BRAM canvas buffer and overwrites every entry with 4'b0000, which is the color value that removes a drawn pixel from being overlaid on top of the camera feed.

B. Center of Mass Stabilization and Stylus Detection

In our top level design, we have created logic to stabilize the *x_com* and *y_com* coordinates to reduce the likelihood of picking up extraneous signals (often caused by our color mask detecting matched pixels elsewhere in the background) as well as logic for stylus detection. Extraneous pixels can cause the *x_com* and *y_com* to jump around on the screen, especially when there is no stylus around to focus on. We use this logic to restrain the center of mass to focus on pixels within a 150-pixel-wide radius to prevent it from drawing if it picks up any random pixels detected by our color mask. We compare the current center of mass coordinates with



Fig. 3. Stamp sprite sheet

the previously calculated coordinates and find the distance between the two points. If the distance between the two points falls outside of our 150-pixel-wide radius, we force the stable center of mass coordinates to remain the previously calculated coordinates rather than updating to the new coordinates. Our draw module only allows for drawing or erasing when the *com_stable* signal is also high alongside the corresponding enable signals. Additionally, we further gated our draw enable signals by making sure that these values are only driven high when the stylus is detected on screen for at least three frames worth of time. By defining an approximate estimate for the size of the stylus tip, We store the number of pixels detected by the mask in each frame, and if it falls within our specified estimate for how many pixels are in our stylus brush for at least three frames, the *draw_en* signal is set high.

C. Stamps

An additional feature we wanted to include with our system were preset stamps. Our draw module takes in two additional signals: *stamp_en* and *stamp_select*. We created a sprite sheet of two stamps: a heart and a fireball using Piskel, an online open-source pixel editor. These stamps were 12 x 12 pixels each, and using the *img_to_mem.py* script we generated the necessary .mem files associated with the stamps. Within the draw module, we added two BROMs (read-only memory) to index and access the 24-bit RGB encodings used in our stamps. However, due to our system's usage of a 4-bit encoding for color, we added logic for color translation to be able to translate the 24-bit RGB encoding into a 4-bit color encoding so that the stamps could effectively be drawn onto the screen (and stored within our canvas BRAM).

We utilize a finite state machine for the stamps to iterate through our stamp BROMs so that we can draw every stamp fully onto the screen. Our FSM for the stamp logic is shown in Fig. 3. The FSM transitions out of the idle state when *stamp_trigger* is high. This occurs when the *draw_en*, *stamp_en*, and *com_stable* signals are high while the *stamp_trigger_prev* is low. This was done to preserve the “stamping” quality during the drawing process; otherwise the stamp would be drawn continuously as long as drawing is enabled in stamp mode. The states STAMP_READ_IMG and STAMP_READ_PALETTE serve as delays due to the image and palette BROMs 2-cycle latency during reads. The STAMP_DRAW state iterates over each pixel in the stamp and transitions back to STAMP-

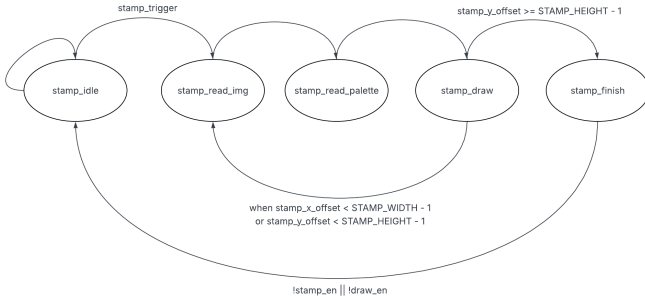


Fig. 4. Stamp finite state machine



Fig. 5. Five fingertips tracked with k-means

READ_IMG at the end of every pixel. Once every pixel has been read, the state transitions to STAMP_FINISH and does not return to idle until drawing or stamp mode has been disabled.

In addition to the stamp drawing, we also included a ‘preview stamp’ that overlays on top of the stylus. This shows a transparent version of the stamp so that users can see where the stamp will be before they place it. Users can switch between our two preset stamps by toggling switch[4]: switch[4] low corresponds to the heart, while switch[4] high corresponds to the fireball.

IV. K-MEANS MODULE (EMMA)

In order to provide an intuitive method for the user to control the brush settings (such as brush width, color, and stamps), we have decided to use a “gesture-controlled” interface, implemented with a glove with three colored foam fingertips.

The process of differentiating between different “gestures” will use a k-means algorithm to approximate the locations of the centroids of the five teal fingertips. The pixels are identified using a sobel mask thresholded to identify teal based on their red chrominance value. The locations of their centroids with respect to each other will allow the interface module to decipher the “gesture” and modify brush settings accordingly.

To differentiate between different fingers raised, our K-means algorithm needs to accommodate k values varying from 0 to 5, deviating from standard implementations of the algorithm. We achieve this by implementing a basic k-means algorithm with assumed $k = 5$, then combinationally pro-

cessing the resulting centroid addresses to determine whether multiple centroids were assigned to one finger. Finally, to avoid losing centroids due to lack of pixels assigned, “dead” centroids automatically get their addresses set near the centroid with the most pixels assigned.

A. K-Means Algorithm

Upon system reset, the centroids are given initial default locations at [213, 360], [426, 360], and [639, 360], [852, 360], and [1065, 360] so that they are approximately evenly spaced across the vertical center of the frame.

For each teal pixel from the mask, the Manhattan distance to each current centroid location is used to identify the closest centroid. The coordinates of that pixel are then utilized to recalculate the new location of that centroid. The choice to use Manhattan distance rather than Euclidean distance allows us to make a trade-off of slightly less accurate distance calculations for speed, since we avoid needing to perform multiplication and square root operations with each calculation. New locations for each centroid are updated after each frame, and for each subsequent frame, the previous frame’s centroid locations are used as the starting centroid value. A standard center of mass calculation is used to determine the x - and y -coordinates of each new centroid. Since the pixel clock speed is much faster than the rate the hand will be moving, using the previous frame’s centroid values allows us to effectively locate the centroid locations of each subsequent frame.

A few adjustments to the general algorithm were made to accommodate for noise and the possibility that less than five fingers were displayed. If the module sees less than 80 teal pixels on the screen, it assumes this is due to noise, sets the total fingers counted to 0, and doesn’t update the centroids for that cycle. In addition, if there are less than five fingers on the screen, the module occasionally will encounter centroids with little to no pixels assigned, often resulting in those centroids getting “lost” as their addresses stop updating. Since “lost” centroids interfere with the final count of fingers in frame, we resolve this by automatically reassigning the addresses of “lost centroids” as slightly offset from the address of the centroid with the most pixels. The reason for using this value is that centroids with the most pixels assigned are either stuck between two fingers or the proper center of a large finger. In the first case, by assigning another centroid to this centroid, with a few iterations of the algorithm, the new centroid will help differentiate the two fingers, making the final finger count more accurate. In the second case, there will just be multiple centroids assigned to one finger, but it will still be counted as one finger due to the hand gesture postprocessing. These modifications optimize a basic k-means algorithm for our purposes.

B. Hand Gesture Postprocessing

In the case that less than 5 fingers are held up, the K-means algorithms will assign multiple centroids to at least one finger. Due to this, we need to add logic to interpret the resulting centroid addresses as the actual correct number of

fingers. We utilize the distance between each centroid and its closest neighboring centroid to determine whether the centroid belongs to a unique finger. This threshold value is controlled by switches 5 through 8, allowing the threshold to be adjusted between the values 8'b00000000 and 8'b01111000. The user can adjust the threshold as needed depending on how close to the camera they choose to use the glove; but we have found that a threshold value of 8'b00111000 or 56 performs well. The module goes through each centroid value, and reasons whether it should be counted as a unique centroid, tallying all the valid centroids to produce live finger count value.

Also, since the threshold values are constantly updating, the live finger count value can be relatively noisy. Therefore, the system only updates the final finger count, and adjusts settings accordingly, once every second. To implement this, we tally the live finger count value every cycle. We do this for 75,000,000 cycles, corresponding to 0.75 seconds, and at the end of the second, the finger count with the highest number of tallies is output as the final number of fingers held up. This value is then passed to the interface module to be displayed and utilized to adjust drawing settings accordingly.

V. INTERFACE (MAYA)

To create a more visually clear and succinct system, we have added a visual interface module that communicates the current drawing settings on screen. This module overlays a graphical user interface on top of the final video output, providing immediate feedback between the screen and the user.

A. UI Interface

The graphics module takes in the control signals produced by the physical switches on the FPGA as well as the hand gesture interpreter (color_sel, brush_size_sel, draw_en, erase_en, stamp_en, stamp_select, and final_fingers). Using these values, the graphics module displays their current settings on the screen. In this way, it acts as a source of visual feedback between the drawing system and the user.

On the screen itself, these signals are translated into three key indicators. (1) An icon that varies depending on what mode is currently being selected. When in draw mode, there is a small circle that reflects both the currently selected brush color and the active brush size. The circle rotates between the 15 different colors (as defined by color_sel) and actively changes size based on which brush width is being selected (as defined by brush_size_sel). This is done such that the user can quickly verify which hue will be applied next and judge the width based off of the circle's diameter. When in stamp mode, the stamp that is currently selected (as defined by stamp_select) will be displayed. When in erase mode, there will be no icon in that area. (2) A small box of text and an icon that corresponds to the mode the system is currently in. This will rotate between DRAW (with a pencil icon), ERASE (with an eraser icon), and STAMP (with a stamp icon) depending on the triggers from the hand gesture interpreter. This provides for clarity showing the current mode, so the user always knows the effect of the stylus before making contact with the canvas.

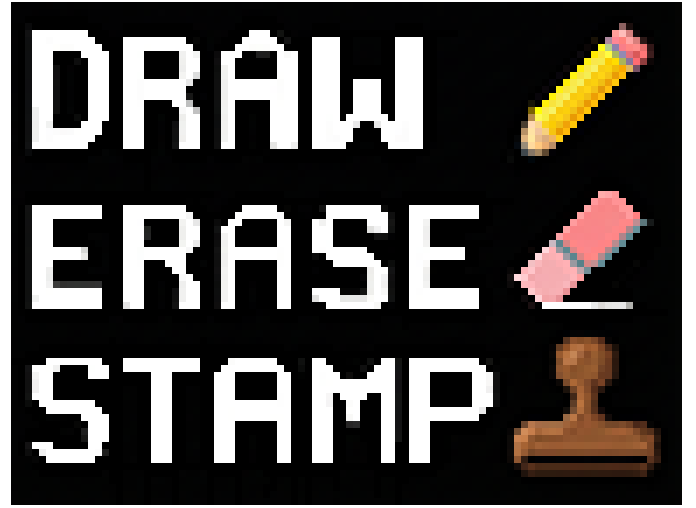


Fig. 6. Drawing Type Sprite Sheet

(3) A “battery” on the top right part of the screen that shows how many “fingers” are currently being recognized by the hand gesture interpreter. Each segment of the battery is colored gray, but it slowly fills up with green as additional “fingers” are recognized (with a max of five segments).brom

Together, these elements form a lightweight visual interface that updates continuously as the user changes settings.

B. Graphics

The specific graphics required for this model meant that we needed to generate four additional BROMs to index and access the 24-bit RGB encoding such that we could easily access both the stamps and the UI overlay. Though the stamp spritesheet from the draw module could be reused, we created an additional spritesheet of three different overlays: draw, erase, and stamp (each with their own custom icon attached). Using the img_to_mem.py script, we then generated the associated image.mem and palette.mem files.

Due to the use of several graphics and calculations needed for the graphics, we had to significantly pipeline our graphics.sv module.

These features significantly improve the usability of the system by allowing the user to understand the drawing configuration at a glance without disrupting their interaction with the stylus or gesture controls.

VI. EVALUATION

Our primary method for evaluating our system's success were the timing and latency of our display output, our total memory usage, and the visual quality of the output we generate on the HDMI display.

A. Timing and Latency

Our system runs on a 100 MHz clock. Within the draw module, it takes 2 cycles to write into our canvas BRAM, another 2 cycles to read from it, thereby only requiring 100 MHz * 4 cycles = 40 nanoseconds of delay to render a drawing

onto the screen, which is virtually imperceptible to the human eye. Stamps are more time consuming, but even then it takes 2 cycles to read into our image BRAM, another 2 cycles to read into the palette BRAM, and then another $12 * 12 * 2 = 288$ cycles to go through the stamping finite state machine. This takes a total of 2.92 microseconds – albeit longer, it is still well under the detectable threshold for the human eye. Our entire system meets timing with a positive WNS=0.219. We had run into issues with negative slack and had to pipeline the top level design and graphics module to remedy it.

B. Memory

The camera buffer is stored in SDRAM at full 1280 x 720 resolution, utilizing $1280*720*16 = 14.746$ MBits out of a total 1 GBit of SDRAM storage. Out of 120 DSPs, we used 9. In Block RAMs, we have 2700 KBits available. In our system, we used:

- Stamp Image Buffer (draw, graphics): $12*12$ byte depth * 8 bit width * 2 stamps * 2 = 4.608 KBits
- UI Image Buffer (graphics): $96*24$ byte depth * 8 bit width * 3 sprites = 55.296 KBits
- Palette Buffers (draw, graphics): $24*256*3 = 18.432$ KBits
- Canvas Buffer (draw): $320*180$ bits/frame * 4 bits/entry = 230.4 KBits
- Total: 308.736 KBits out of 2700 KBits

C. Visual Quality

Overall, the visual quality of our system works well. Our user interface provides visual feedback on the current action mode (erase, draw, stamp) alongside information about the brush (color and size) using sprites that change according to whatever settings have been selected for the brush. We are able to use and switch between all 16 colors, stamp both stamps in their entirety, and erase or clear the screen of any drawings. Furthermore, the K-means algorithm is effective in tracking and detecting how many fingers are displayed on screen, allowing for easy brush setting selection and accomplishing one of our primary goals for this project. We stabilize the fingertip detection by only allowing mode changes if the K-means algorithm tracks the fingertip(s) for at least 0.75 seconds. This reduces the amount of noise that might come from spurious pixels in the background.

D. Checklist

We were able to successfully accomplish our commitment goals, as a hand gesture interface was implemented using color thresholding and we have working preset stamps and brush settings that are selectable using hand gestures. We made some progress towards custom stamps, however there were issues in the address calculation as only a small portion of the drawing would be captured and during stamping, the captured stamp preview did not match the stamp that was actually placed down. The final step to fleshing out this project would be transitioning away from color thresholding and towards feature detection for the hand gesture interface.

Some ideas for additional use cases for our project, which include:

- Additional stamps; since most of our switches have been freed up thanks to the hand gesture interface, we could add more stamps for users to select from.
- More colors; our color selection is controlled via hand gestures, so adding more colors would simply involve adding additional cases to our color selection logic.
- Highlighter brush; introducing new brush types that have a semi-transparent stroke

VII. IMPLEMENTATION INSIGHTS

Throughout our project, we learned some valuable lessons:

- Admittedly, color thresholding can be finicky. Different lighting conditions can affect the performance of our system in terms of the k-means finger tip detection as well as drawing. One lesson learned from this is to not use a color that is close to human skin tones for the drawing stylus, otherwise the user needs to wear long sleeves and gloves (and maybe a mask) to interact with our whiteboard. Having a masking layer and cross hairs were incredibly helpful in debugging.
- Due to the visual application of our system, we spent a lot of time analyzing the visual impacts of changes to our modules but would often forget about the physical hardware timing constraints inherent to FPGAs. In the future, if working on a visual FPGA product, it would be useful to pay attention to both pipelining and effectiveness in the beginning rather than having to deal with all the timing constraints at the end.
- Also, in the future, rather than solely using one threshold for the mask, it could be useful to threshold for with various color channels to more accurately filter for specific colors and eliminate noise.
- Our project was very modular; we had done a good job in designing how each piece interfaces together. This was incredibly beneficial in enabling each of us to work on our portion of the project individually and it was relatively easy to integrate at the end when connecting all of our modules together.

VIII. CODE REPOSITORY AND INDIVIDUAL CONTRIBUTIONS

Our code repository containing our full system is found here: <https://github.mit.edu/6205F25/fa25-6205-team09>

A. Contributions

Eileen designed and implemented the draw module. Eileen also drew the stamps used for the preset stamps. She also contributed to the integration effort to iron out the relationship between the fingertip detection and the onscreen GUI. With respect to the report, she drew up the high-level diagram and wrote the abstract and high-level summary as well as contributing to the evaluation section and implementation insights.

Emma designed and implemented the k-means module and hand gesture postprocessing logic. She integrated the glove setting control into the draw and graphics interface. In the report, she contributed to the evaluation and implementation insights sections.

Maya worked on the graphics interface. She designed and implemented the GUI interface and pipelined the top level design and graphics design to ensure that the project met timing constraints. She helped integrate the k-means module and the onscreen GUI. She designed and 3D printed the Mickey Mouse stylus for the system. She also put together and edited the final video and contributed to the implementation insights.

IX. ACKNOWLEDGMENTS

We would like to thank Prof. Joe Steinmeyer and Monica Chan, alongside the rest of the 6.205 teaching staff for all their guidance in this project.