SpeedyAR

Max Katz-Chrsity
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

Miles Kaming-Thanassi
Dept. of Electrical Engineering and Computer Science
Massachusetts Institute of Technology

***Abstract*—We present a design for an augmented reality system implemented on a NEXYS 4 DDR FPGA. In this system, a virtual object is projected into the camera frame which remains in place relative to the motion of the camera. We utilize HSV object tracking to generate three vectors from a set of four tracking points. A virtual object is projected into the scene using a linear combination of these vectors. We are able to animate this virtual object by shifting between different sets of virtual points in a set number of frames.**

I. Introduction

This paper outlines a design and implementation of an augmented reality system. Augmented reality allows for mixing the artificial and real world by fixing virtual objects in real world camera feeds using sensor readings. Typically, AR systems identify some known objects in the real world and fix virtual objects to them providing the illusion that the objects present in the real scene. A common application of AR is to fix a virtual piece of furniture within a room in order to simulate what the piece would look like in real life.

Traditional methods for creating AR utilize the perspective-n-point algorithm to calculate the camera's rotation and direction relative to a set of known tracking locations. This algorithm requires an understanding of complicated linear algebra which we found difficult to accurately implement in hardware. We decided to circumvent the complexities of this traditional approach by generating a set of vectors from for four tracking points and then representing a virtual object as a linear combination of these vectors. The accuracy of the linear combination strategy is lowered as the virtual points get further from the tracking points. This leads to distortion if we try to display points further from the tracking points, because we do not properly account for the perspective projection of the camera image, and assume that it is orthographic. A link to the final project code can be found here: https://github.com/mileskt/SpeedyAR_Final_copy.git.

II. Object Tracking

In order to generate the set of vectors with which we can project a virtual object into the camera frame, we first needed to determine the location of a four tracking points. Since we needed to track something invariant to

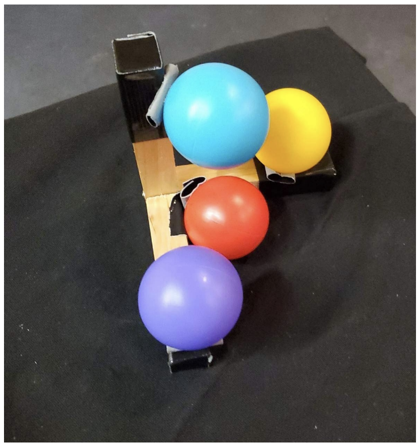

Fig. 1. The setup we built to hold each of the tracking balls

rotation of the camera, we used color to distinguish the tracking objects from each other and the rest of the scene. Effectively tracking color in the RGB color space output by our camera was difficult because it varied drastically with light. To overcome this, we used a converter, written by Kevin Zheng in 2010 to transform the RGB output of our camera to the HSV color space where each pixel is represented by a hue, saturation and value. We encountered difficulty when implementing this converter because the IP divider originally used is out of date, so we had to figure out how to tweak the IP generator to get a divider with the proper latency and ratio of numerical to fractional output bits. For testing, we defined a set of thresholds using the switches on the FPGA and then displayed a mask over all the pixels within this threshold. we determined thresholds that accurately isolated each color with minimal noise. We used a ring light to get an even distribution of light over the balls.

III. vector generation

The masks output by each of the HSV threshold modules are fed to a center of mass calculator that finds the average x and y location of each of the tracking objects. This tracking data is then fed into a module which can calculate the vectors used for projection. The x and y vectors are represented with packed 48 bit arrays divided into three 16 bit sections representing the distance of each tracking point from the origin. We chose 16 bits so that the vector values can be converted to a fixed point

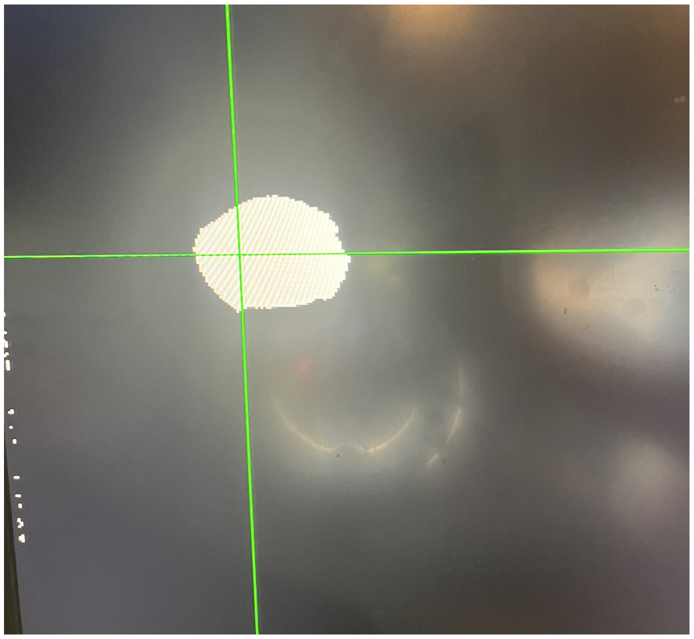

Fig. 2. Center of mass tracking using HSV filtering on the color purple

representation using an arithmetic left shift. The vector module is a state machine with two states outlined below:

- Idle: In the idle state, the module waits for new center of mass values for each of the tracking objects. Once all of the center of mass values are known, the module will switch into the calculating state.
- Calculating: In the calculating state, x, y and z vectors are calculated. In order to calculate each vector the locations of the purple, green and blue balls are subtracted from the center of mass of the red ball. This gives us a vector system centered around the red tracking ball.

IV. virtual points

The virtual object that we project into the camera frame is represented by a set of points that we outline below:

A. *Point structure*

Each virtual point is 52 bits wide and has the following structure:

Color	z	y	x

- Color: The top two bits of the virtual points are used to represent color. The color indicates what the color line will be drawn from that point. A color value of zero indicates that no line should be drawn from the point. Due to memory constraints we could only use three colors which we chose to be red, green and yellow.
- Z,Y and X: The remaining 48 bits of a virtual point represents three 16 bit scalar values. Each value is represented in fixed point, two's compliment notation with eight integer and eight floating point bits. These scalars are use to scale the x,y and z vectors generated from our tracking points.

B. *Point Generation*

We wrote a Python script to generate each of these points so that the resulting mesh would look like a block character (specifically mimicking Minecraft's Steve character). To be able to display the animated object, the set of points and colors need to be loaded into the BRAM of the FPGA. Three dimensional objects are stored in a number of file formats, such as Wafefront's `.obj` file and the Standard Triangle Language `.stl` and the more primitive Polygon File Format `.ply`. The general idea of these files is all the same, storing some combination of points, lines, polygons, and surfaces with various amounts of specificity regarding color and other formatting. Before moving to animations, we created a few tools to convert theses object files into a set of ordered points that would draw out an object on the FPGA, however, we found theses files to be relatively inefficient as they relied not on a continuous line but on a series of loops with often overlapping edges. Additionally, with animation, there are few files available that fit our specifications, so we generated the file from a Python script.

The script has a set of objects that are able to draw themselves, and a parent object that draws all of it's children objects. Each independently moveable object is based around its own coordinate system with a rectangular prism and optional additional features, translated and rotated from the global origin pose. The rotation matrices for each object are given by sine functions over time, allowing them to swing back and forth smoothly. Stepping through a set of times, we can draw the points at each location. All the points can be converted into linear combinations of the vectors between the known tracking objects, which is easy when the unit vectors are chosen, and written to memory file in hex. Concatenating all the frames together gives us a memory file that can be flashed to the device.

C. *Animations*

In order to animate the virtual objects, we rotate through different sets of virtual points that contain the desired object in different positions and orientations. This is handled by the load_animations module. We store all sets of virtual points in a single port read only BRAM which is read from at an offset value that rotates every 300 frames.

V. Point Projection

A key module in our system is the projection module which takes the position of the red tracking point as the origin and the x,y and z vectors calculated in the vector

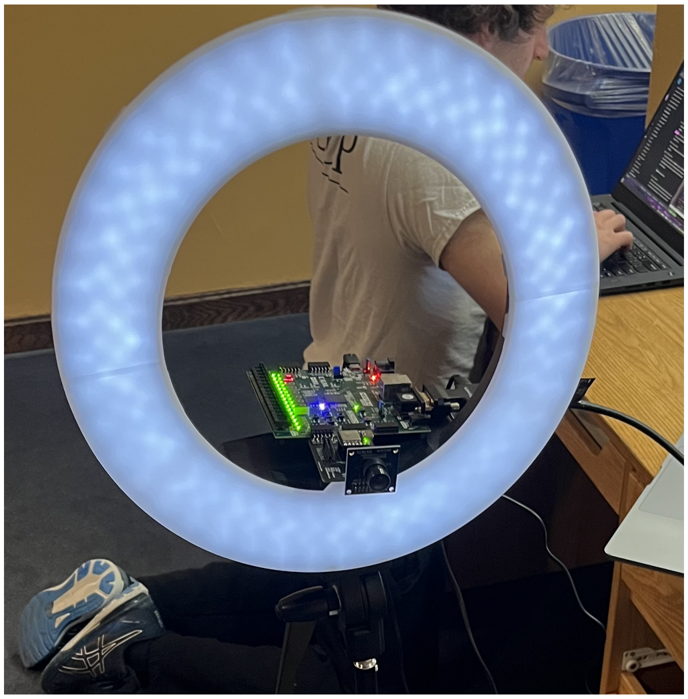

Fig. 3. The physical setup of the FPGA in the center of a ring light.

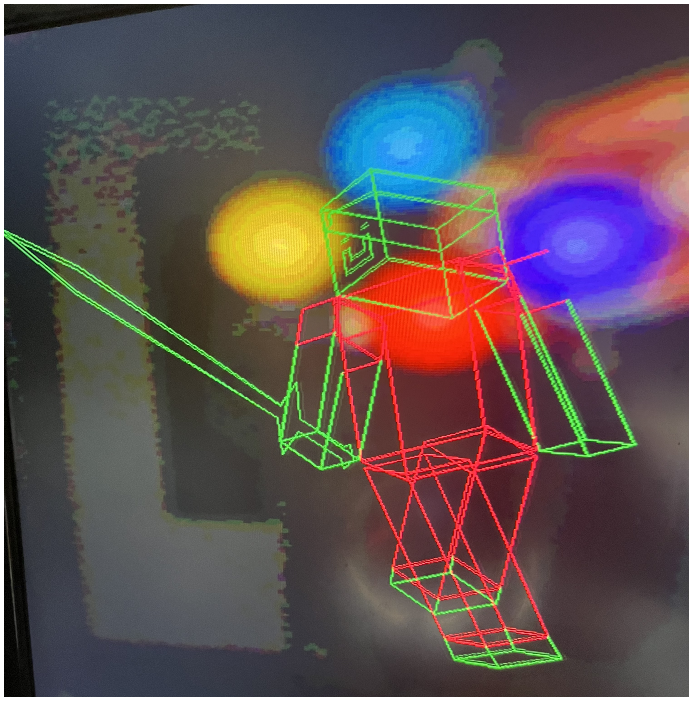

Fig. 4. block character projected into the camera frame using tracking objects.

module. for each of the virtual points being projected, we compute a linear combination of the vectors and add back the value of the psuedo-origin coordinate. Additionally, All values in this module are represented using two's compliment to support negative directions. The projection module is pipelined so that each projected point is calculated and output sequentially.

Our original stated goal was to implement a pose estimation algorithm to calculate the camera's 6 degrees of freedom. We spent some time working on this, looking at different solutions to the perspective n-point problem. The algorithms we looked at used a set of known tracking points and their locations in the 2d plane as well as the camera's intrinsic properties to estimate the location and roation of the camera. From the camera's pose, the virtual points can be accurately displayed in the camera frame using accruate and deterministic projections. There are many different algorithms to solve the perspective n-point problem, each with a balance between accuracy and complexity. Unfortunately, even simpler algorithms such as P3P and EPnP were still quite complex and given the timing constraints and the depth of the math required to understand the implementation. After failing to implement and understand these algorithms, we decided to devise a simpler process for point projection. This is what brought us to the very simple yet often innacurate linear combination of known vectors strategy.

Although this method of projection is much simpler than the pnp algorithm, it still produced many bugs due to the nuances of fixed point, two's compliment arithmetic. Additionally, we encountered situations were the math would work in Iverilog test benches but not in Vivado. Specifically, We found that left shifting the vector values to convert them to fixed point worked in test benches, but in Vivado would cause overflow. This discrepancy made it extremely difficult to determine the source of our errors and cost us much valuable time that could have been channeled into improving the project. We fixed this problem by padding the vector values with eight bits on the right and carefully shifting the results of the scalar multiplication to ensure that only the integer portion of each operation would be output. We also store the output our projection calculation into a 32 bit value to avoid incorrect truncating until all calculations are complete. An example of our final projection calculation can be seen below:

```
x_projection <= ((($signed({green_x_vec,8'b0})*
    ↪ $signed(point_scalars[0]))>>>16␣+((
    ↪ $signed({blue_x_vec,8'b0})*$signed(
    ↪ point_scalars[1]))>>>16)+(($signed({
    ↪ purple_x_vec,8'b0})*$signed(point_scalars
    ↪ [2]))>>>16))+␣$signed({1'b0, x_origin});
```

VI. Line Drawing

In order to create mesh objects, we implemented Bresenham's Algorithm to draw lines between the projected points output by the projection module. This algorithm takes starting and ending x and y values and sequentially interpolates intermediate pixels between them. The latency of Bresenham's algorithm is proportional to the length of the line being drawn, so we use a flag to tell the projection module when to feed it a new point. The line drawing module has two states which we outline below:

- Idle: In the idle state, we set the previous end point as the new start point and set the most recent input from the projection module as the next end point. It is important to note that once after every reset we have to wait for an initial point from the projection module to be set as the start. In addition to setting the start and end points, the module also calculates the difference between the start and end points, and the direction of the line to be drawn. After these initial calculations are complete, it will switch to the calculating state.
- Calculating: In the calculating state we sequentially calculate the location of every point in the line being drawn. Once the calculated x and y position is equal to the ending x and y position, we switch back to the IDLE state and set a flag telling the projection module to send the next point out.

VII. Pixel Management

Storing and managing all of the projected points proved to be difficult, so we implemented a separate module just for handling pixel management. Having a distinct module for pixel management also gave us the freedom to easily iterate on our pixel management scheme as the project progressed. The inputs to this module are the current *h_count* and *v_count* as well as the current projected point and its color output from the projection module. The pixel manager reads the stored pixel value at *h_count* and *v_count* from one BRAM and writes the new projected pixel values to another. In order to read and write simultaneously, we needed to add a third BRAM to clear stale points. The pixel manager rotates the function of each of the three BRAMs at the end of every 480x640 frame.

The writing BRAM takes as input the color of the projected point at the current x and y location. We found that writing to an address outside of the bounds of the BRAM leads to wrapping, so we ignore all values that lie outside the bounds of the camera frame. Similarly, the reading module ignores all *h_count* and *v_count* values that are out of bounds. The clearing BRAM simply writes a zero to the current *h_count* and *v_count* location.

We found the pixel management module very difficult to test because it could not be synthesized in iverilog due to the BRAM IPs. In order to ensure that the BRAMs were being properly rotated and storing the correct values we had to spend a hours testing the module manually on the FPGA and carefully evaluating the module by hand using state diagrams.

VIII. System Utilization

A. Latency and Throughput

- Vector Generator: one cycle of latency, a new set of vectors is calculated once per frame.

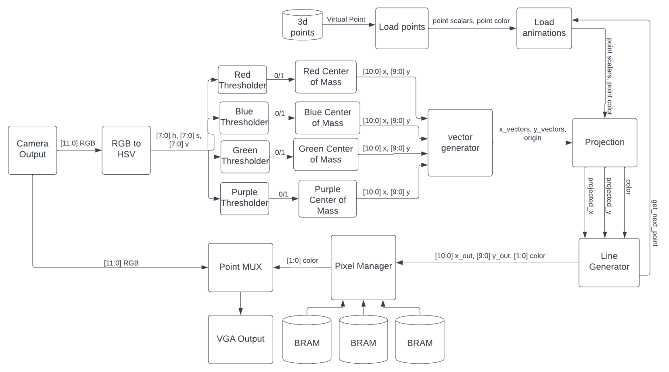

Fig. 5. System block diagram

- Load animations/Load Points: two cycles of latency to read from BRAM. Throughput is not a useful metric for this module because it only outputs a new point when the next_point input flag is high.
- Projection: There is one cycle of latency and the throughput is one point per cycle.
- Line Generator: This module calculates one point in the line per cycle, so the latency is proportional to the length of the line being drawn. We are not drawing lines in parallel, so the throughput is also proportional to the length of the line.
- Pixel manager: 2 cycles of latency due to reads from BRAM.

B. Memory Utilization

The overall memory utilization for our project ended up being much greater than we expected. According to usage reports, we used 115 out of 135 available block RAM tiles for a total utilization of 85%. Our memory usage primarily comes from the following two places:

- Virtual points: The projected virtual character is represented by 202 points. In order to create animations, we use 10 different sets of points for a total of 2020 virtual points. The `.mem` file containing these points is loaded into BRAM. Each virtual point has a width of 52 bits, so the total size of the virtual point BRAM is 105040 bits.
- Pixel manager: The pixel manager utilizes three separate BRAMs. Each of these BRAMs has a width of 2 bits to represent color, and a depth of 640x480 for a total of 1843200 bits. This size proved to be a bottleneck for adding color support which we had to shrink from 4 bits to 2 to avoid resource utilization errors.

C. Miscellaneous Utilization

The remaining utilization of our FPGA was relatively low compared to memory. We used only 5% of LUTs, 30% of I/O and 10% of DSPs.

IX. Retrospective

A. Challenges

While we initially thought color tracking would be the easiest part of the project, it ended up being one of the most difficult. Converting to HSV was tricky in and of itself, because we had to learn how to use the Xilinx IP divider and the Vivado graphical interface. We also often got different completely different threshold results because of variations in lighting despite HSV giving us more lenience with the ranges. We finally got tired of manually testing different values each time we wanted to test with the camera in different lighting, so we invested in a ring light to place around the FPGA. The ring light drastically stabilized our results. Through this process, we kept improving our system for experimenting with thresholds, building internal tooling that would allow us to efficiently produce the final values.

We spent a significant amount of time chasing down bugs that were from order of operations and typing details in our logic's. The worst bugs were the ones hidden in `top_level.sv`, as these could not easily be test benched. We realized over the course of the project the importance of having small modules with strong hierarchy that complete very specific and isolated tasks so that everything can be test benched. Because our project is significantly display and camera based, we had to be creative with creating test benches where we could see a detailed output. We also had to be very careful using BRAMs as these caused issues for test benching. Once we were able to replicate the issues in the test bench, we were quick to find and fix the actual issue in hardware. We were surprised how much time we had to spend isolating small seemingly simple sections of code, but it payed off in the end.

B. Evaluation

The results of the project accomplished our overarching goals. We did have a major pivot partway through, allowing us to bypass some of the intermediate goals, but we utilized the simplicity of our final method, to extend our project into a couple of our reach goals like animations. We would have liked to implement a perspective-n-point algorithm for estimating camera pose as this is an interesting research problem, but we didn't realize how tricky it would have been to implement, so we had to abandon this to complete the project. After pivoting, we decided to focus on stabilizing the tracking system, getting accurate projections and adding additional features like animations and color. We initially set out to only project a point cloud onto the scene, so we are very happy with the graphics work we did to create final animated block character.

C. Further Work

Given more time to work on our project, we would have liked to improve the tracking through methods beyond color that might be more reliable. We discussed using beacons that flash at different frequencies and patterns. This would stabilize the resulting animation and allow for the camera to move more freely. It would also allow us to move the camera further from the beacons, making it a more robust and useful system. There also other tracking systems to consider, such as pattern matching as is done in professional film. We also think it would be interesting to build a more mobile camera setup that could rotate around the tracking objects more like traditional AR.

As mentioned previously, it is possible to implement perspective-n-point algorithms on the FPGA, and this would allow for the projection onto the real world to be perspective instead of the less accurate orthographic projection we currently have. This would be particularly important when the animated object needs to be a large distance from the tracking object or a different depth than the tracking objects.

The wireframe is good for showing accurately where the object is placed, however it isn't very realistic, and we would have liked to implement surfaces and shading. This would require some work to figure out which planes are hidden, as well as efficiently storing all of the object skins. Additionally, it would be helpful to explore memory efficiency changes that could allow us to display a larger set of colors.

X. Author Contributions

Over the course of this project we frequently found ourselves working together on each of the modules and test benches. The idea for an augmented reality project was Max's initially, so he did much of the initial planning and brainstorming work. Miles worked on the line drawing module and Max wrote the python mesh generation script, but the remaining modules were for the most part written together. We spent much time during the project debugging and writing test benches. It was useful to do this testing work together because we could bounce ideas off one another and avoid getting stuck on sneaky bugs. Overall we feel that we put equal effort into the project and are happy with how we meshed as a team.