Xilinx FPGA Augmented Reality Cards

6.205 Class Project: Final Report

Omozusi Guobadia
Department of Electrical Engineering & Computer Science
mozig@mit.edu

Ezra Kang
Department of Electrical Engineering & Computer Science
ehk@mit.edu

***Abstract*—This project is a Xilinx FPGA-Camera system that works in tandem with developed AR Cards to produce 3D images that appear on the monitor within the actual environment. While moving the camera, in addition to seeing the normal display of the environment, the user should be able to view various angles of the 3D image on the monitor.**

I. Introduction & Motivations

In 1968, Ivan Sunderland, a computer scientist at Harvard University, developed the Sword of Damacles, known to date as the first augmented reality head-mounted display system [3]. Since then, there have been multiple companies that constructed devices that integrate AR to create more interactive and immersive content.

A popular, successful instance of this technology was introduced by Nintendo. In 2011, the company released a system compatible tool in which users can utilize their 3DS to view Augmented Reality figures on their system. Cleverly identified as AR Cards, these items interfaced with the on-system 3DS camera, for the user to freely capture interactive displays of Nintendo Characters of their choice, and also allowed individuals to creatively add them to select games on the system [1].

Our project in a similar way aims to implement our very own AR Cards with the Xilinx FPGA-Camera system model. Our model will track the various custom made AR cards distinctively, and depicts different 3D models depending on the design of the card.

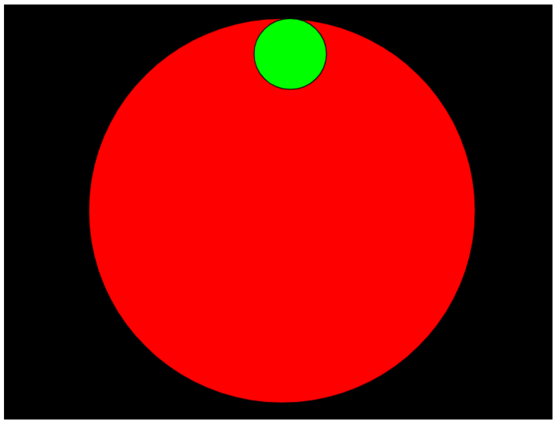


Fig. 1. Image of an AR Card

The AR Cards will be tracked using distinctive color coded marks on the edges as well as the middle of the card, while the 3D Modeling of the actual image will be made utilizing a vertex-rasterization pipeline, which is similar to Interactive Minecraft project constructed by Alexis Camacho and Chenkai Mao [2]. Depending on the view of the camera, a different angle will be shown of the 3D image. Due to the speed at which the FPGA computes its frames, and the intrinsic parallel nature of Verilog in its storage of values and computing functions, this is almost a perfect device to execute this task on.

II. Setup

The FPGA-Camera system will be set up on a tripod at a specified distance to limit the variability in the design. The setup will also include a white mat for the card to be blacked on that ignores noise in other objects, as well as a square AR Card that will be a color between red, blue, and green. The setup of the FPGA-Camera is initially projected to be at an angle at sixty degrees,derived from the accelerometer information from the FPGA system itself, and a distance forty centimeters away from the center of the mat. An image of the projected system is provided in Fig. 2, as well as full image depiction in Fig. 3 and Fig. 4.

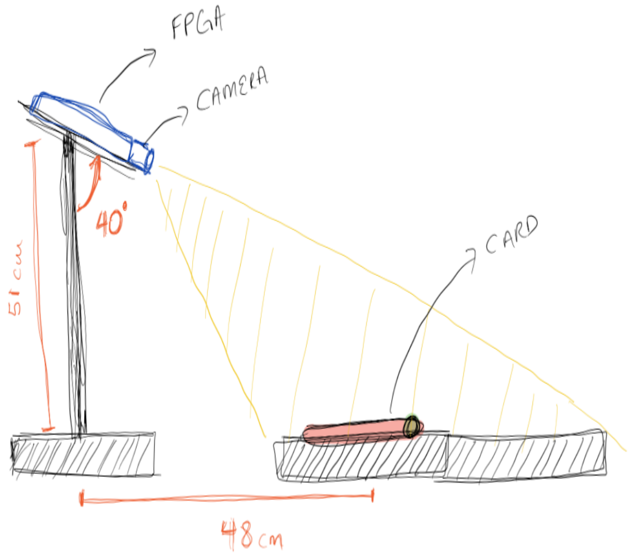


Fig. 2. Proposed Setup of Xilinx-FPGA Camera

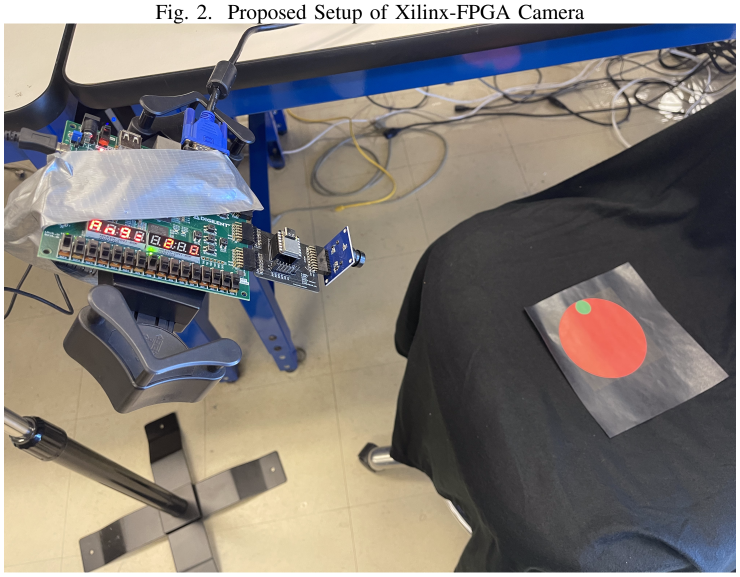


Fig. 3. 1st POV Setup of Xilinx-FPGA Camera

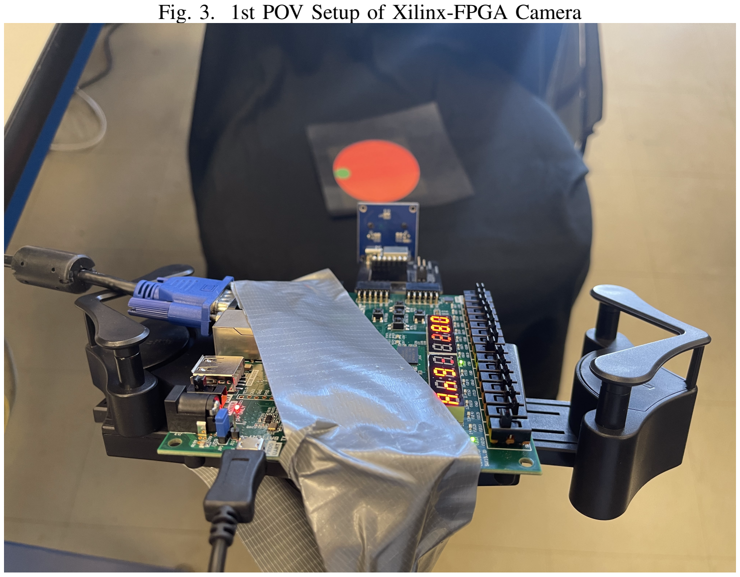


Fig. 4. 3rd POV Setup of Xilinx-FPGA Camera

Fig. 5. Setup of Xilinx-FPGA System

III. Approach (Modules)

The preliminary design is broken into three key processes in order for the AR Card to function: the Pre-Rendering, Rendering Pipeline, and the Post-Rendering Pipeline. The Pre-Rendering Pipeline will be utilized to obtain the regular camera raw output, and rotate the output in order to receive hcount and vcount information, or x and y infomation that is easier to code into modules (this was an arbitrary decision to include before or after the rendering pipeline). The Rendering Pipeline will use essential functions to create the image perceived onto the monitor, replacing the regular card image partially. The Post Rendering Pipeline outputs essential information as the center and angle at which the card is positioned, which will then be recycled to be used in the Rendering Pipeline itself. These modules will be described in full detail.

A. Pre-Rendering Pipeline

The Pre-Rendering Pipeline is broken up into two modules, the camera module which obtains the camera raw output, as well as the rotate module, which converts the axes in which the raw output for the camera is perceived from y-x axis (which is vertical information being on the horizontal, and horizontal information being on the vertical axis in comparison to the monitor screen), to x-y axis (which is the exact opposite, and what people are used to). The clocking for this pipeline will be sixty-five megahertz, which is essentially the maximum capability of the attachable FPGA Camera. The Clock Generation Module produces the necessary frequency needed for the Camera attachment to function efficiently. The VGA generation module produces the sync information for the pixel display on the monitor, as well as provides a way to obtain regular x and y information for the whole system without being limited by specific modules or adding extraneous variables in the system itself. Written Explicitly

- **Clk Gen**: This module will be included to produce the sixty-five megahertz clock frequency for the camera and vga, based on the normal FPGA clock frequency. Accepts 100MHz as input and returns sixty-five megahertz as output (1 Bit).
- **Camera**: This module handles the camera information, accepts the signals that generally come from the camera board and returns the output frame done flag (1 Bit), pixel valid out flag (1 Bit), and pixel out data (16 Bits).
- **Rotate**: Rotates the image in the correct orientation and returns the pixel address (12 Bits), when given the frame done flag (1 Bit) as the input.
- **Recover**: Retrieves the (16 Bits) camera output at 65MHz.
- **VGA GEN**: Handle Synchronization information in terms of frame modeling. Outputs the hsync (10 Bit), vsync (9 Bit), hcount (10 Bit), and vcount (9 Bit) information.

B. Rendering Pipeline

The distance and the angle information will be provided for the camera, to which we programmed have been tested with a phi (spherical coordinates) of forty degrees and an of sixty-four centimeters. The theta and camera coordinates will determine the image of the picture composed and seen. These variables can be input into a rendering pipeline to generate a model on screen in real-time. The goal is to use the camera's perspective, along with a 3D model defined by its vertices, to output certain pixel colors at the correct position on the monitor. There are several stages to this process.

- **Model Representation**: A 3D model is traditionally represented as an array of vertices in 3D space, with texture mapping to put color onto the model. The first step of the rendering process would break up these vertices into triangles defined by its three vertices, which is a shape that is both robust in terms of representing an object well, and efficient in terms of the calculations necessary farther down the pipeline. In the interest of managing complexity, we will forgo the step of calculating triangles from vertices and instead represent a model by its triangles directly. Texture mapping is a difficult problem, early computer graphics would instead give a single color to each triangle, which is the approach we take here. Specifically, a model is stored in a BRAM, initialized with lines of sixty-four bits. Each line contains a single triangle, with the x, y, and z coordinates of its three vertices, and a ten bit color. Each coordinate is a signed six bit number, giving the model a resolution of sixty-four.
- **From 3D triangles to 2D triangles**: The first step of the rendering pipeline is to take the camera position and a line from the model BRAM, and perspective shift the triangle such that it is translated from the 3D space to a 2D space. This allows the triangle to be displayed on a 2D screen in a way that looks correct from the camera's perspective. To accomplish the perspective projection, the triangle's vertices need to be defined by the camera's coordinate system, where the origin is the camera's position and the axes are aligned with the angle from the model's origin. Note that in the camera's coordinate system, the X and Y axes correspond to horizontal and vertical positions on a screen, and the Z axis corresponds to how far away the vertex is from the camera depth-wise. The first calculation is getting the vector from the camera's position to each of the vertices. This is done with a simple vector subtraction. Then it is necessary to rotate the coordinate system, which is accomplished by multiplying the vectors by a rotation matrix The first step of the rendering pipeline is to take the camera position and a line from the model BRAM, and perspective shift the triangle such that it is translated from the 3D space to a 2D space. This allows the triangle to be displayed on a 2D screen in a way that looks correct from the camera's perspective. To accomplish the perspective projection, the triangle's vertices need to be defined by the camera's coordinate system, where the origin is the camera's position and the axes are aligned with the angle from the model's origin, as shown in the figure. Normally, this is done by taking

the inverse of a four-by-four matrix that can translate a point defined in world space to coordinates in the camera space. Calculating this matrix, taking its inverse, and multiplying the coordinates of a point by that inverted matrix is a very computationally intensive process. A fair amount of assumptions are made to significantly reduce the complexity of this problem. Examples of these assumptions are that the camera has a fixed angle in the z-axis, the model is assumed to be viewed from at least a certain distance away, and the model is always assumed to be in full view.

The sines and cosines are calculated with the sine-table module, which accesses a BRAM of precomputed sine values. This BRAM contains ninety fixed point sixteen-bit numbers, corresponding to ninety degrees of resolution of sine values. The sine-table module can take in a theta of range [0, 360] and calculate the correct sine value with combinational logic to determine the right BRAM address to lookup. The cosine can also use this sine table, as cos(theta) is the same as sin(theta+90). The BRAM has a latency of two cycles, but is fully pipe-lined so both of the required values can be received in three cycles.

The final step is to perspective project the triangle, which is achieved by dividing the X and Y coordinates by depth (Z). Normally, after the perspective projection the pixel coordinates are reduced down to a range of [-1, 1] (as decimals). The coordinates can then be scaled accordingly. However, a fixed point division in hardware would double the amount of cycles needed compared to a non-fixed point one. This problem was worked around by first multiplying the X and Y coordinates by their scale (32 in this case, which has the extra advantage of allowing bit shifts rather than real multiplication), chopping off the decimal bits, and then they are divided. The bit-shifts add two cycles to the division, but with the aforementioned method there would have been four cycles added, along with a more costly fixed point multiplication. Division is costly compared to bit shifts, but it is necessary here to preserve the accuracy of the image. At this point, the coordinates are fourteen bits to preserve information from the various calculations. Note that the coordinates are rounded to integers before the division, because this information is not very important as there are no half-pixels, and reducing the bit length reduces the division complexity. The division takes six cycles, but it is fully pipe-lined. After the division of the three vertices is complete, the X, Y, and Z coordinates can be passed to the rasterize module, along with the color. The overall latency of the module is ten cycles, but it is almost fully pipe-lined, ensuring that the module is efficient. This module is fully pipe-lined with the caveat that if the downstream rasterization module is processing a triangle, and the 3dto2d module is ready to output, then the pipelines within this module are paused. The rasterization process takes a variable amount of cycles depending on the triangle that is passed in, so it must output a busy signal to indicate that it cannot receive any inputs in that cycle. However, even if the rasterization module is busy, the 3dto2d module still continues its calculations as long as it is not currently ready to output a triangle. This ensures that this module still makes efficient use of its cycles.

- **Rasterization**: Rasterization is the process of taking a triangle defined by three vertices, and calculating which pixels lie within the triangle so it can be colored correspondingly. There are several techniques to accomplish rasterization, but we chose edge computation as it is relatively simple to implement and still efficient to calculate in terms of latency. Edge computation requires using the three vertices to make three functions that take in a X and Y coordinate. If these three functions all evaluate to a positive number, then that coordinate is within the triangle. An edge function takes the form E=Ax+By+C. There are three pairs of vertices: (v0, v1), (v1, v2), (v2, v0). These pairs are used to find the constants for Equation 1.

$$\begin{aligned} A &= Vy_i - Vy_j \\ B &= Vx_j - Vx_i \\ C &= Vx_i * Vy_j - Vx_j * Vy_i \end{aligned} \tag{1}$$

Now we must test each pixel to see if it fulfills the equations. It would be inefficient to check every possible pixel coordinate, so we use the bounding box technique to narrow down the search. The minimum and maximum X and Y of the three vertices are found. No pixel outside these bounds can be in the triangle, so only the range of pixels within these bounds are tested. One pixel is tested per cycle. If the pixel passes the test, then that pixel's coordinates and color are sent to the z-buffer module. One pixel per cycle seems inefficient at first, as a triangle spanning its whole possible area means that the rasterize module would take around four thousand (specifically, 4096) cycles to complete just one triangle. However, the rasterize module does not end up being the bottleneck in the overall pipeline. The end goal is to write pixel colors into a BRAM, and at full throughput for a BRAM is limited to a single write per cycle. The issue with the current approach is that there is an appreciable chance that there will not be full throughput. If a pixel fails the edge test, then a cycle was wasted not sending a value to be written to the BRAM. There are two ways to reduce the chances of a cycle being wasted. The first is to calculate tighter bounds of the bounding box (such that it is more of a bounding polygon.) The second is to test multiple pixels in the same cycle. A combination of both would be best, and we will work towards implementing this.
- **Z-Buffering**: In the case that a pixel of a triangle was behind another triangle, we do not want to display that pixel. This means that the depth of that pixel must be checked. This information was already calculated further up the

pipeline. Note that what was stored was the depth of each of the three vertices of the triangle, not necessarily the depth of each pixel within the triangle. Calculating this is possible with interpolation, but it would add unnecessary complexity. The only cases where interpolation would matter would be when part of a triangle is behind another triangle, and another part is in front of that same triangle. To reduce complexity, it can be assumed that the model is well-formed, so that this situation does not happen. Then the depth that can be used is the smallest depth of the three vertices of the triangle. The z-buffer module takes in an X, Y, depth (Z) and color. It accesses two BRAMs, with one containing the depth of the pixel at a location and the other containing the color of the pixel at a location. The address of a pixel location is calculated from the X, Y, and then a read is sent to the BRAM containing depth data. If the depth returned is more than the depth of the current pixel the module is looking at, then this pixel color should be replaced (the current pixel is closer to the camera than the old pixel.) Therefore a write request is sent to both BRAMs to replace the color and depth at that location. If the depth returned is less, then nothing should be written. After the whole model is processed, there is now a BRAM containing all of the correct pixel color data. The reason why two BRAMS are used is because it allows for full throughput, meaning that there can be reads and writes in the same cycle. If one BRAM was used to store both depth and color, then the z-buffer module can only read or write in a cycle (as this BRAM is also used by the render module.) Adding another BRAM allows the z-buffer module to read/write to the depth BRAM in the same cycle, write to the color BRAM, and still have one port open for the render module.
- **Rendering**: The final step in the rendering pipeline is to output a pixel color to the display. According to the hcount, vcount, and image base location, the render module picks the correct pixel from the two pixels (the raw camera output pixel and model color pixel) and outputs it to be displayed over VGA to the monitor.

C. Post-Rendering Pipeline

The Post Rendering Pipeline computes two necessary details for the system. The first is that it computes which pixel information to send out to the overall monitor, which is tabulated by the VGA Mux at the end of the pipeline. The second is that the pipeline also computes the angle and center of mass information needed for the rendering pipeline, utilizing the angle guesser module as well as the center of mass module. The angle guesser module functions by counting the mass of each horizontal line that is seen by the FPGA camera to be within the AR Card, and predicts the angle based on the previous tabulated angle as well as the perceived tilt of the card itself. The center of mass module works by similarly counting the mass number of pixels the camera perceives to be included within the AR Card, and divides the summed location in both the x and y direction by the total mass. More Explicitly:

- **Framebuffer**: Retrieves the pixel color information (16 Bit) when given the address of the pixel.
- **Address Picker**: Sends the appropriate pixel address information (16 Bit) when given the current hcount (11 Bit) and vcount (10 Bit) information.
- **Scale**:Scales the size of the output imposed image by 8/3. Returns pixel color output (16 Bit) information.
- **Threshold Detector**: Returns true if the pixel color output meets a certain threshold value.
- **Center of Mass (Card Body)**: Returns the corresponding center of the card body, x-com(11 bit) and y-com(10 bit). Dependent on the threshold value.
- **Center of Mass (Angle Detector)**: Returns the corresponding center of the card body, x-com(11 bit) and y-com(10 bit). Dependent on the threshold value returning true.
- **Seven Segment Converter**: Returns to the FPGA board corresponding values to create the LED Number Display, cat-out(7 bit) and an-out(8 bit).
- **Angle Guesser**: Returns the angle guessed (9 bit) based on the center of mass information of the card body and the angle detected.
- **Angle to Coordinate Converter**: Returns the x (9 bit signed), y (9 bit signed), z(8 bit) information for rendering based on the angle information, as well as a predetermined distance specified away from the card.

D. Full Approach

The system starts by perceiving a card. The initial cycle bypasses the rendering pipeline until angle and center of mass is derived. After the information is calculated, the information will be sent to the rendering pipeline for further processing, in which after the image is rendered the pixel information will be taken to the vga-mux to be seen by the user on the monitor. A full block diagram of the project is included at the end of the report.

IV. Evaluation/Results

The rendering pipeline starts processing when it receives a valid camera location. The latency from a valid signal to the first pixel being written is fourteen cycles. If the overall latency were to be quantified by the amount of cycles required to process a whole model, then the latency would depend on both the amount of triangles in the model, and the size of those triangles. For a maximally large triangle (where its vertices range from [-64, 64]) the rasterization takes around four thousand cycles. Therefore the overall latency for processing a maximally large model is equal to (4096+13)*(amount of triangles in model).

The throughput depends on how it is quantified. If the throughput is defined by the pixels being written to the BRAM, then the pipeline has full throughput (a pixel is written per cycle at full throughput). If instead the throughput is defined by its processing of valid signals, then it does have full throughput. The pipeline pauses when the rasterization module is currently processing a triangle and another triangle is ready to be sent to it.

The rasterization takes the most amount of time in the pipeline. In a more average case, with a triangle of size sixteen, the rasterization still takes around two hundred (specifically, 256 cycles) cycles to complete. However, the rasterize module does not end up being the bottleneck in the overall pipeline. The end goal is to write pixel colors into a BRAM, and at full throughput the BRAM is limited to a single write per cycle. However there is still room for optimization here. The issue with the current approach is that there is an appreciable chance that there will not be full throughput. If a pixel fails the edge test, then a cycle was wasted not sending a value to be written to the BRAM. There are two ways to reduce the chances of a cycle being wasted. The first is to calculate tighter bounds of the bounding box (such that it is more of a bounding polygon.) The second is to test multiple pixels in the same cycle. A combination of both would be best.

The biggest improvement to the throughput and latency of the system would be changing the RAM that is used such that multiple writes are allowed per cycle. Overall the RAM usage of the rendering system is three BRAMs, one for sine values of size (90 * 16 bits), one that holds the pixels of the model (64*64*10bits), and one that holds depth information for z buffering (64*64*9bits). These sizes could technically be reduced, but that would mean decreasing the robustness of the design (by having less accurate sine values, or a smaller resolution of the model.)

The timing requirement is the pixel clock speed, sixty-five megahertz. The WNS of the synthesized design is three nanoseconds, which makes sense considering our design implementation worked effectively, since it is borderline near being a terrible design, it wouldn't't make sense to fit more logic in per cycle.

In terms of DSP utilization, since we only utilized the camera, it makes sense that the overall utilization is six percent. Since we now know this, the potential opens up for utilizing different cameras to process varying AR cards. For instance one camera will be dedicated on processing red output information and the other camera will be dedicated on processing blue output information, and we could superimpose the two depictions to generate the two AR images in a single frame. We could also use a different function like audio to signal the AR image to jump and become more interactive, to add additional features.

The total memory utilization was even smaller than anticipated. The BRAMs, the major component in terms of memory, only took up twenty-eight percent of the total memory usage. This means that we also have the potential to add up to two more AR images to create specificity in card choice. Of course, this means we would have to create two more cards to exercise this feature. We could also have a more detailed AR image encoded, which would take up more memory. LUT utilization was kept below ten percent, although this was not a huge concern for us.

V. Discussion/Retrospective

A. Kang's Retrospective

There were several lessons we had learned throughout the development of this system. The rendering pipeline processes and moves a large amount of bits per cycle. In the interest of reducing bit bleed, we wrote the modules to use unpacked arrays. However, IcarusVerilog has problems working with unpacked arrays, so the modules that used unpacked arrays could not be simulated. Instead, we tested the modules on hardware by outputting onto the seven segment display. This ended up taking an extremely large amount of time, both in waiting for compilation and having to write less effective tests. In hindsight, it would have been better to duplicate the modules to use packed arrays so that they could be tested in simulation, but use the unpacked versions on hardware. The process of perspective projecting the 3D triangles into 2D space ended up being a much larger and complex problem than first anticipated. In fact, this module was as complex as the rest of the rendering system put together. Because this was not understood beforehand, a substantial amount of time was spent having to replan and rewrite this module. This part of the project would have benefited greatly from properly understanding the requirements and better planning. The rendering pipe-lining ended up involving an abundance of computer graphics concepts. We had no prior knowledge of the subject beforehand, and this was a major hindrance throughout the project. Particularly with respect to the perspective projection, modules had to be heavily modified or rewritten when it turned out that the math was incorrect. In the future, heavy guidance should be sought for topics that one is not necessarily expected to know for the class, such as computer graphics.

B. Guobadia's Retrospective

Over the course of programming the pre-rendering pipelines and post-rendering pipelines, I gained a new found appreciation for the 3DS, especially for its complex processing hardware for handling AR Card Detection. Especially for it to be able to process AR Cards in a variety of different environments irrespective of changes in background color.

This was a two component project, in which I had to think of the optimum environment for tracking the AR Card, and also creating the code to be able to run the detection (for both the center of mass and the angle detection). To limit the amount of noise introduced in the system, which was especially concerning considering that there was no way of filtering out the noise entirely, we had to use a black cloth as the background. Looking back, it would have been ideal to realize this early, as it would have limited the amount of hours dedicated to detecting the card. I also designed the card to be as simple as possible, as in the beginning, I overestimated the cameras detection abilities (I initially thought that the camera would be able to detect color information at the span of a mm, but the camera's resolution capacity is way less). So even though this was a coding project, this also fit under the description of a “design” project.

The code needed to detect the card needed a lot more requirements from the video pipe-lining template lab. We needed to optimize the code, change unneeded sequential parts to combinatorial, to maximize the speed of processing the angle and center of mass.In addition to adding the angle detector modules (I also needed a way to reduce noise that did not include the filter modules; the filter modules had additional BRAMs that would have sent the memory utilization upwards), this dedicated a majority of the time for my section of the project. Although the ideation process was very insightful, it would have been great if I could skip, so it would have been also to implement other stretch goals for the project.

Another idea to create a robust system, with more AR Designs and Card Detection capabilities that we could have implemented, was to try to integrate multiple FPGAs to double the memory and processing capabilities. Although that would have been an additional amount of weeks dedicated to proper integration. Maybe a component for the class in the future would be to integrate into one of the labs, how to use multiple FPGAs on the same project (it could be a sub-project lab to get used to working in teams before the final project).

VI. Author Contributions

A. Guobadia's Contribution

I looked into optimizing the video processing pipeline in order to reduce the amount of memory utilized, including an angle detector pipeline. I also designed and made the card, and the environment needed for card detection. I was involved in proof-reading the project documents, restructuring the block-diagrams and images, and researching augmented reality, its history and general tips for creating an AR system.

B. Kang's Contribution

I researched computer graphics extensively in order to understand how a full rendering pipeline is implemented. This included the full problem of going from a digital model defined by vertices all the way into outputting pixels onto the screen in correct positions. Additionally, I had to solve several problems related to the assumptions and optimizations that one can make in order to simplify the rendering system. Overall, I researched and implemented the entire rendering pipeline. I also did quantitative analysis on the system in terms of timing, latency, throughput, and resource utilization. I wrote the portions of the report and created figures relating to the rendering pipeline.

References

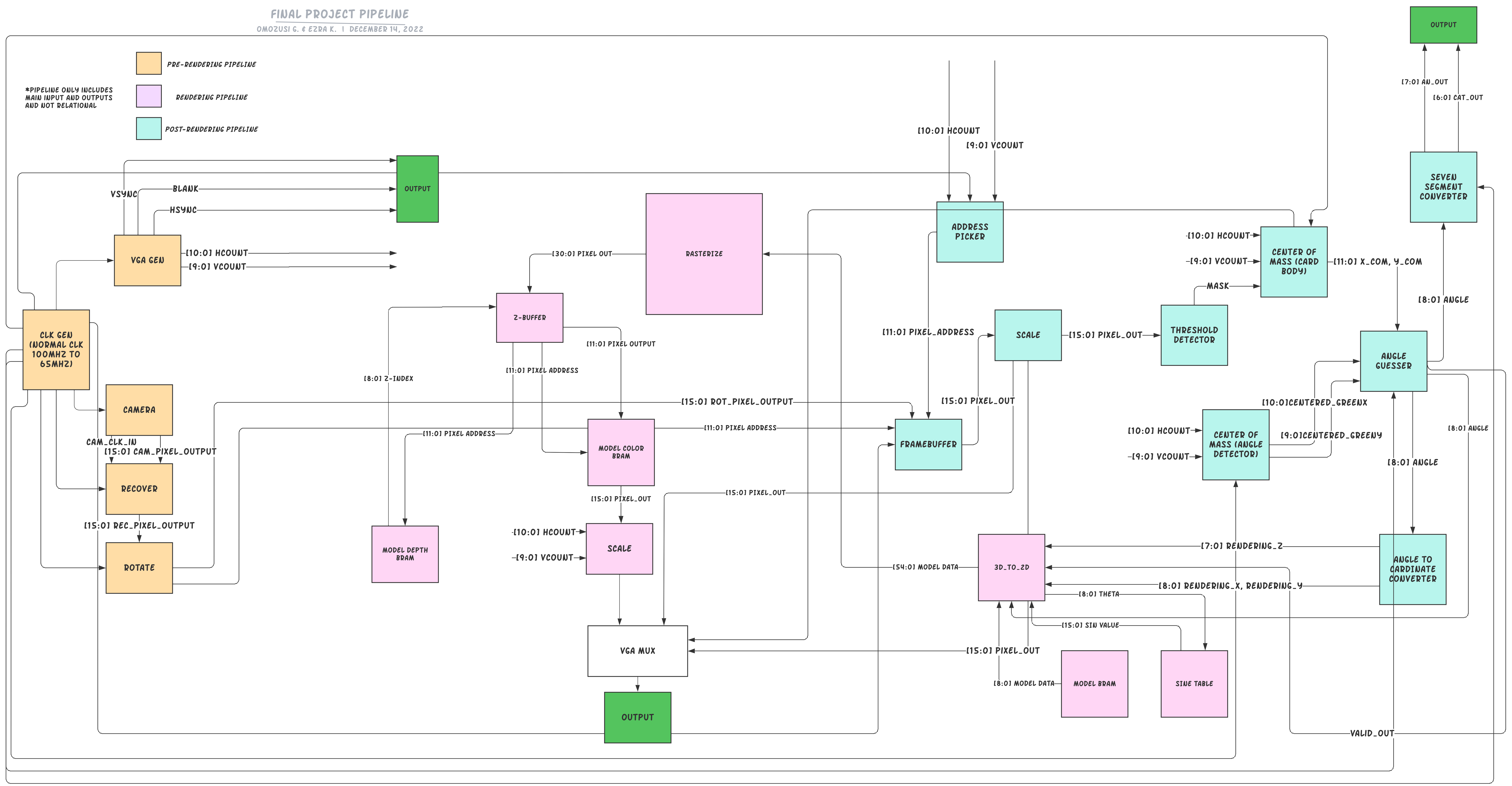
[1] AR cards. Nintendo. (n.d.). Retrieved October 25, 2022, from https://nintendo.fandom.com/wiki/AR_Cards

[2] Camacho, A., Chenkai, M. "Interactive Minecraft" MIT, Digital Electronics Laboratory, Fall 2020.

[3] The mainstreaming of augmented reality: A brief history. Harvard Business Review. (2016, October 4). Retrieved October 25, 2022, from https://hbr.org/2016/10/the-mainstreaming-of-augmented-reality-a-brief-history: : text=The%20first%20AR%20technology%20wa s,AR%20head%2Dmounted%20display%20system.

FINAL PROJECT PIPELINE

OMOZUSI G. & EZRA K. | DECEMBER 14, 2022



```
Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------------
| Tool Version : Vivado v.2022.1 (lin64) Build 3526262 Mon Apr 18 15:47:01 MDT 2022
| Date         : Wed Dec 14 17:44:22 2022
| Host         : EECS-DIGITAL-27 running 64-bit Ubuntu 20.04.5 LTS
| Command      : report_utilization -file /tmp/tmp.KugjIu/obj/placerpt_report_utilization.rpt
| Design       : top_level
| Device       : xc7a100tcsg324-1
| Speed File   : -1
| Design State : Physopt postPlace
------------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Slice Logic Distribution
3. Memory
4. DSP
5. IO and GT Specific
6. Clocking
7. Specific Feature
8. Primitives
9. Black Boxes
10. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+------+-------+------------+-----------+-------+
|          Site Type         | Used | Fixed | Prohibited | Available | Util% |
+----------------------------+------+-------+------------+-----------+-------+
| Slice LUTs                 | 3874 |     0 |          0 |     63400 |  6.11 |
|   LUT as Logic             | 3791 |     0 |          0 |     63400 |  5.98 |
|   LUT as Memory            |   83 |     0 |          0 |     19000 |  0.44 |
|     LUT as Distributed RAM |   28 |     0 |            |           |       |
|     LUT as Shift Register  |   55 |     0 |            |           |       |
| Slice Registers            | 2336 |     0 |          0 |    126800 |  1.84 |
|   Register as Flip Flop    | 2318 |     0 |          0 |    126800 |  1.83 |
|   Register as Latch        |   18 |     0 |          0 |    126800 |  0.01 |
| F7 Muxes                   |    7 |     0 |          0 |     31700 |  0.02 |
| F8 Muxes                   |    0 |     0 |          0 |     15850 |  0.00 |
+----------------------------+------+-------+------------+-----------+-------+


1.1 Summary of Registers by Type
--------------------------------

+-------+--------------+-------------+--------------+
| Total | Clock Enable | Synchronous | Asynchronous |
+-------+--------------+-------------+--------------+
| 0     |            _ |           - |            - |
| 0     |            _ |           - |          Set |
| 0     |            _ |           - |        Reset |
| 0     |            _ |         Set |            - |
| 0     |            _ |       Reset |            - |
| 0     |          Yes |           - |            - |
| 0     |          Yes |           - |          Set |
| 19    |          Yes |           - |        Reset |
| 1     |          Yes |         Set |            - |
| 2316  |          Yes |       Reset |            - |
+-------+--------------+-------------+--------------+


2. Slice Logic Distribution
--------------------------

+-------------------------------------------+------+-------+------------+-----------+-------+
|                 Site Type                 | Used | Fixed | Prohibited | Available | Util% |
+-------------------------------------------+------+-------+------------+-----------+-------+
| Slice                                     | 1436 |     0 |          0 |     15850 |  9.06 |
|   SLICEL                                  |  971 |     0 |            |           |       |
|   SLICEM                                  |  465 |     0 |            |           |       |
| LUT as Logic                              | 3791 |     0 |          0 |     63400 |  5.98 |
|   using O5 output only                    |    2 |       |            |           |       |
|   using O6 output only                    | 2462 |       |            |           |       |
|   using O5 and O6                         | 1327 |       |            |           |       |
| LUT as Memory                             |   83 |     0 |          0 |     19000 |  0.44 |
|   LUT as Distributed RAM                  |   28 |     0 |            |           |       |
|     using O5 output only                  |    0 |       |            |           |       |
|     using O6 output only                  |   28 |       |            |           |       |
|     using O5 and O6                       |    0 |       |            |           |       |
|   LUT as Shift Register                   |   55 |     0 |            |           |       |
|     using O5 output only                  |   31 |       |            |           |       |
|     using O6 output only                  |   15 |       |            |           |       |
|     using O5 and O6                       |    9 |       |            |           |       |
| Slice Registers                           | 2336 |     0 |          0 |    126800 |  1.84 |
|   Register driven from within the Slice   | 1478 |       |            |           |       |
|   Register driven from outside the Slice  |  858 |       |            |           |       |
|     LUT in front of the register is unused |  380 |       |            |           |       |
|     LUT in front of the register is used  |  478 |       |            |           |       |
| Unique Control Sets                       |   56 |       |          0 |     15850 |  0.35 |
+-------------------------------------------+------+-------+------------+-----------+-------+
* * Note: Available Control Sets calculated as Slice * 1, Review the Control Sets Report for more information regar
ding control sets.


3. Memory
---------

+-------------------+------+-------+------------+-----------+-------+
|     Site Type     | Used | Fixed | Prohibited | Available | Util% |
+-------------------+------+-------+------------+-----------+-------+
| Block RAM Tile    |   39 |     0 |          0 |       135 | 28.89 |
|   RAMB36/FIFO*    |   38 |     0 |          0 |       135 | 28.15 |
|     RAMB36E1 only |   38 |       |            |           |       |
|   RAMB18          |    2 |     0 |          0 |       270 |  0.74 |
|     RAMB18E1 only |    2 |       |            |           |       |
+-------------------+------+-------+------------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1


4. DSP
------

+----------------+------+-------+------------+-----------+-------+
|    Site Type   | Used | Fixed | Prohibited | Available | Util% |
+----------------+------+-------+------------+-----------+-------+
| DSPs           |   15 |     0 |          0 |       240 |  6.25 |
|   DSP48E1 only |   15 |       |            |           |       |
+----------------+------+-------+------------+-----------+-------+


5. IO and GT Specific
---------------------

+-----------------------------+------+-------+------------+-----------+-------+
|          Site Type          | Used | Fixed | Prohibited | Available | Util% |
+-----------------------------+------+-------+------------+-----------+-------+
| Bonded IOB                  |   76 |    76 |          0 |       210 | 36.19 |
|   IOB Master Pads           |   35 |       |            |           |       |
|   IOB Slave Pads            |   38 |       |            |           |       |
| Bonded IPADs                |    0 |     0 |          0 |         2 |  0.00 |
| PHY_CONTROL                 |    0 |     0 |          0 |         6 |  0.00 |
| PHASER_REF                  |    0 |     0 |          0 |         6 |  0.00 |
| OUT_FIFO                    |    0 |     0 |          0 |        24 |  0.00 |
| IN_FIFO                     |    0 |     0 |          0 |        24 |  0.00 |
| IDELAYCTRL                  |    0 |     0 |          0 |         6 |  0.00 |
| IBUFDS                      |    0 |     0 |          0 |       202 |  0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |          0 |        24 |  0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |          0 |        24 |  0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |          0 |       300 |  0.00 |
| ILOGIC                      |    0 |     0 |          0 |       210 |  0.00 |
| OLOGIC                      |    0 |     0 |          0 |       210 |  0.00 |
+-----------------------------+------+-------+------------+-----------+-------+


6. Clocking
-----------

+------------+------+-------+------------+-----------+-------+
|  Site Type | Used | Fixed | Prohibited | Available | Util% |
+------------+------+-------+------------+-----------+-------+
| BUFGCTRL   |    3 |     0 |          0 |        32 |  9.38 |
| BUFIO      |    0 |     0 |          0 |        24 |  0.00 |
| MMCME2_ADV |    1 |     0 |          0 |         6 | 16.67 |
| PLLE2_ADV  |    0 |     0 |          0 |         6 |  0.00 |
| BUFMRCE    |    0 |     0 |          0 |        12 |  0.00 |
| BUFHCE     |    0 |     0 |          0 |        96 |  0.00 |
| BUFR       |    0 |     0 |          0 |        24 |  0.00 |
+------------+------+-------+------------+-----------+-------+
7. Specific Feature
-------------------

+-------------+------+-------+------------+-----------+-------+
|  Site Type  | Used | Fixed | Prohibited | Available | Util% |
+-------------+------+-------+------------+-----------+-------+
| BSCANE2     |    0 |     0 |          0 |         4 |  0.00 |
| CAPTUREE2   |    0 |     0 |          0 |         1 |  0.00 |
| DNA_PORT    |    0 |     0 |          0 |         1 |  0.00 |
| EFUSE_USR   |    0 |     0 |          0 |         1 |  0.00 |
| FRAME_ECCE2 |    0 |     0 |          0 |         1 |  0.00 |
| ICAPE2      |    0 |     0 |          0 |         2 |  0.00 |
| PCIE_2_1    |    0 |     0 |          0 |         1 |  0.00 |
| STARTUPE2   |    0 |     0 |          0 |         1 |  0.00 |
| XADC        |    0 |     0 |          0 |         1 |  0.00 |
+-------------+------+-------+------------+-----------+-------+


8. Primitives
-------------

+------------+------+---------------------+
|  Ref Name  | Used | Functional Category |
+------------+------+---------------------+
| FDRE       | 2316 |        Flop & Latch |
| LUT3       | 1743 |                 LUT |
| LUT4       | 1419 |                 LUT |
| LUT2       | 1028 |                 LUT |
| CARRY4     |  913 |          CarryLogic |
| LUT6       |  518 |                 LUT |
| LUT5       |  268 |                 LUT |
| LUT1       |  142 |                 LUT |
| SRL16E     |   64 |  Distributed Memory |
| OBUF       |   47 |                  IO |
| RAMB36E1   |   38 |        Block Memory |
| IBUF       |   29 |                  IO |
| RAMS32     |   28 |  Distributed Memory |
| LDCE       |   18 |        Flop & Latch |
| DSP48E1    |   15 |    Block Arithmetic |
| MUXF7      |    7 |               MuxFx |
| BUFG       |    3 |               Clock |
| RAMB18E1   |    2 |        Block Memory |
| MMCME2_ADV |    1 |               Clock |
| FDSE       |    1 |        Flop & Latch |
| FDCE       |    1 |        Flop & Latch |
+------------+------+---------------------+


9. Black Boxes
--------------

+----------+------+
| Ref Name | Used |
+----------+------+
```

10. Instantiated Netlists

```
+----------+------+
| Ref Name | Used |
+----------+------+
```