6.111 Final Project Report
Extended Sight with EOG

Crystal Wang and Elizabeth Mittmann

Contents

1 Introduction

An Electro-Oculogram (EOG) is a device which measures the position of the retina along with whether the eye is open or closed using electrodes. These electrodes can be placed in pairs above and below or to the sides of the eye. Since the eye acts as a dipole, as one looks around, the potential difference between the pair of electrodes changes accordingly. Thus, from an EOG the wearer's eye movement can be recorded.

For our 6.111 project we created a functioning EOG and output display. This display has several modes, selectable through an overlaid menu which is scrollable through eye movement. The first mode is "data visualization" which shows the state of the user's eyes as determined from the EOG data. The next is the "real world" in which you can switch between multiple cameras, mounted on motors, each of which can be panned based on your eye movement, through selecting a specific portion of the camera feed and through servo movement. Finally, there is a "virtual world" which one can look around in as well, though this was not fully successfully implemented.

One concern was how to ensure that the user could see the screen while controlling the camera's view. To do this we implemented an incremental movement system. If the user wants to pan the camera to the right they can simply look right and then look back at the screen - the degree of turning depends on the length of time they're looking right.

2 Overview

The design of our EOG relies on electrodes measuring the differential voltage up-down and left-right across the eyes. Both of these differences are then amplified so that they are readable once put into the Nexys4.

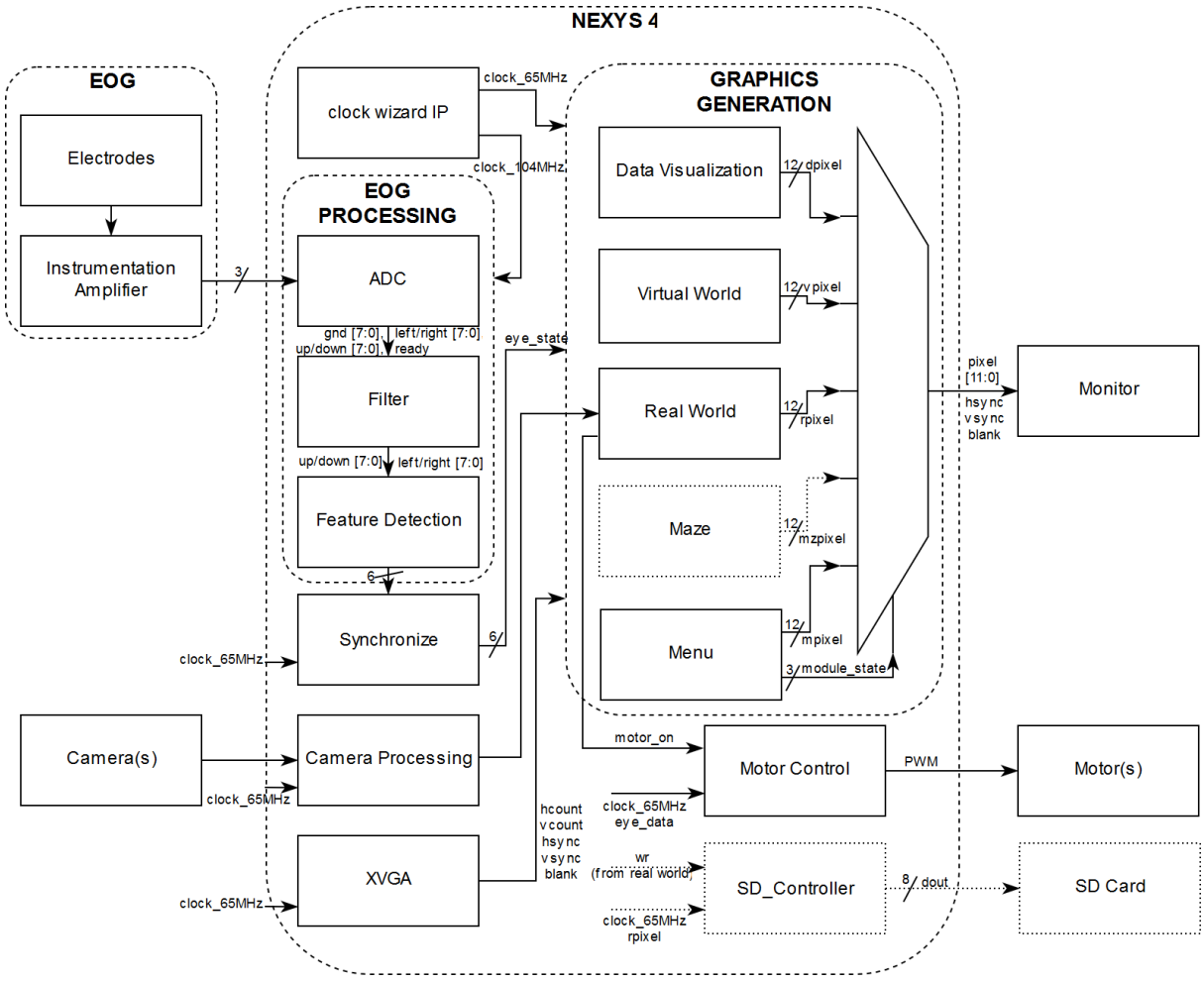
The next group of modules is EOG Processing which consists of reading the analog signals into a digital format in the ADC and then filters these signals so that feature detection can be run on them. EOG processing will output variables indicating the direction the user is looking in or if they are blinking for an extended period of time.

From here the next group of modules is Graphics Generation where a menu (opened by a double click) switches between data visualization, the "real world", and the "virtual world". This large block is also connected to the cameras (through a camera processing modules), and the servos the cameras are mounted on.

- Data Visualization: In this environment cartoon eyes are displayed on the screen, which mimic the actions of the user.
- Real World Environment: In this environment the user can choose between viewing two external cameras on the monitor. As the user moves their eyes the image displayed on the monitor will move to show what is in that direction (i.e. if the user looks up the monitor image will pan up slightly).

We also mounted both cameras on servos to rotate as the user looks left or right.

- Virtual World Environment: In this environment a pre-generated image will be used instead of a live camera feed. This pre generated image will allow the user a 360 degree ability to view the virtual world, as the image will wrap back on itself in a sphere-like format.
- Menu: To switch between these monitor display options of eyes, real world, and virtual world, we implemented a menu which the user can enter with a double click. In the menu the user can scroll (again with eye movements) through the display options, and select one to view.

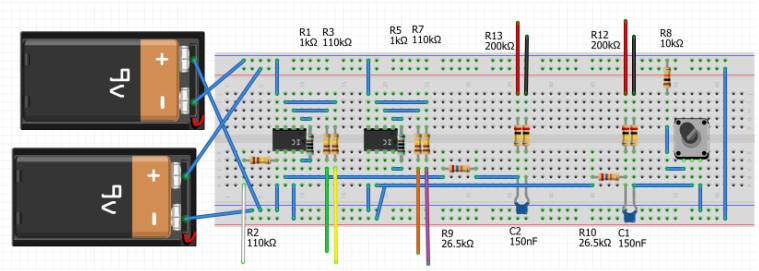


3 EOG Analog Circuitry (Elizabeth)

Much of the analog circuitry was modeled of 6.169's fall EKG lab. This was then adjusted for an EOG using other sample EOG projects found online such as:

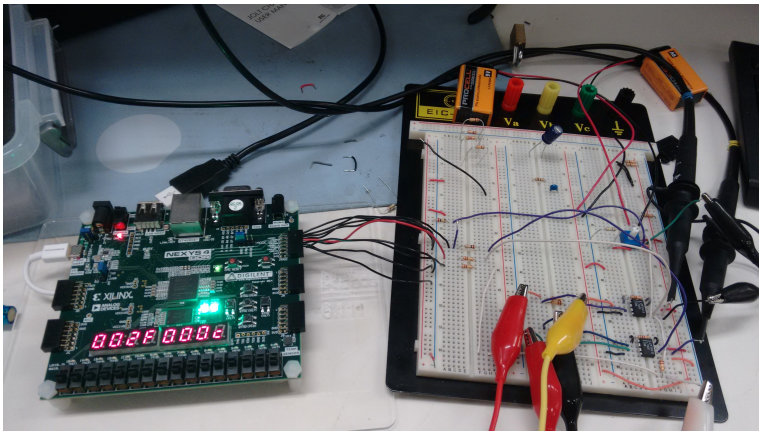
- http://onloop.net/hairyplotter/
- http://eeghacker.blogspot.com/2013/11/measuring-eog-with-my-eeg-setup.html?m=1
- https://www.ecnmag.com/article/2010/04/analog-front-end-design-ecg-systems-using-delta-sigma-adcs

The final circuit was:



IC: LT1920 Instrumentation Amp
Potentiometer: R 103 122 C (10k ohm)
Input: green (right), yellow (left), orange (top), purple (bottom), white (forehead)
Output: left wires (left/right differential), right wires (up/down differential)

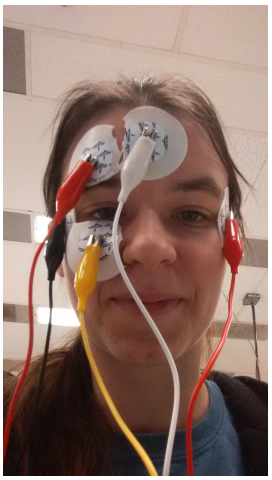
Which in real life looked like:



3.1 Electrodes

I used Foam ECG Electrodes Ag/AgCl, Stress, solid gel (ref: MDS611930A) for sensing voltage differences up/down and left/right across the eye. I used a new set of 5 electrodes each time and connected the electrodes with alligator wires to the circuit. Any point on the circuit which connected to an alligator clip and electrode first was connected through a 110k ohm resistor for the wearer's safety. In addition, I powered the EOG analog circuitry from two 9V batteries to limit the supply of current to the system - again for user safety.

The motivation behind this choice of electrodes and my knowledge of batteries and resistors for safety comes from the 6.169 EKG project, whose leftover electrodes I used. The electrodes were placed on the face as shown below, with corners ripped off so they fit on my face. I placed electrodes on my face 10+ times throughout the project and Crystal put them on her face a couple times. We did not run into any electrode placements where up/down and left/right could not be detected, however we did find that if the electrodes were unstuck and restuck they were likely to become loose and not work.

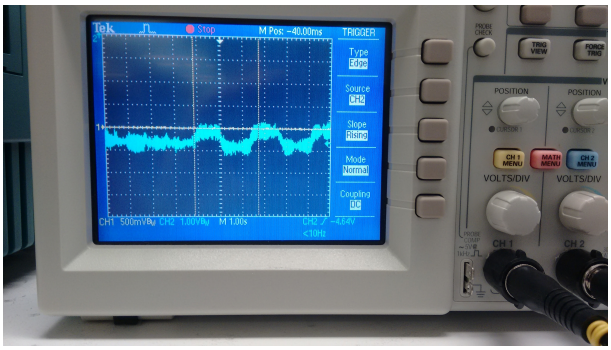


Example electrode placement

3.2 Instrumentation Amplifier

I used two AD620 Amplifiers to amplify the up/down differential with respect to forehead ground and the left/ right differential with respect to forehead ground. These amplifiers were powered by two 9V batteries in series so they had +9V for their positive input and -9V for their negative input (again used in that manner as it had functioned well for the 6.169 EKG). I initially started with a gain of 500 as it was recommended by EOG projects I found online, which made the differential easy to see, however the absolute max was then higher than 1V and was not a possible input to the nexys4. I ended up using a gain of 50 as this

created a signal that stayed within a reasonable range. (See the next section for a further discussion of attempts to use circuitry instead to eliminate DC offset.)



Unfiltered up-down signal with a gain of 500

3.3 DC Offset

I ended up simply using a resistor and potentiometer controlled DC offset for when the amplified signals were not between 0V and 1V as was required by the Nexys4 for input. Each time before connecting the inputs to the Nexys4 I would probe each amp's output and adjust the DC offset. As much as possible I left these probes on during testing as an added check that I was not exceeding the Nexys4's input abilities.

3.3.1 Other Attempts

I attempted to use a capacitor as a high pass filter to eliminate this DC offset, however I found that I would need a large capacitor to create a DC offset which was average over a long enough period of time to not just eliminate the DC offset that was my signal. However, the capacitors that were this size were polar, meaning sometimes I would be violating their labeled polarity.

I next tried using a simple resistor divider as an offset, however I found that with each day and each new person the DC offset varied - for example, most days my left-right differential was centered on 0V, however the day of my checkoff it was centered at .5V. I tried to figure out if I could use an opamp as a subtractor, to normalize the signal with the time averaged signal through the capacitor and then apply a DC offset, however I didn't get this to work either and ended up deciding to simply lower the amplification until the signal could easily fit between 0 and 1 volt, and then use a DC offset which I fine tuned each time with a potentiometer.

3.4 Low Pass Filter

Given limited BRAM space, and the fact that eye signals occurred in the range of 1 - 30 Hz, I decided to only sample data from the XADC at a rate of 500Hz.

However this meant that the higher frequency noise could not be filtered out digitally, so on the output of each amplifier I added a simple low pass filter with a 26.5k ohm resistor and a 150nF capacitor for a cutoff frequency of 40 Hz.

4 EOG Processing (Elizabeth)

4.1 ADC

The ADC took in the differential amplified analog signals from electrodes coming out of the low pass filter, along with the system clock and alternated between reading each of the signals. This was slightly harder to implement than anticipated as the example I was able to download online was from an older version of Vivado than the one I was running, and thus I was unable to open or view the XADC IP. To read each differential I used the Nexys4's ability to read two probes at once, and read one probe that was ground, or a DC voltage offset, and the other probe that was my signal. The module checked every clock cycle to see if the input had been fully read, and on the done signal, swapped over to reading the other pair of probes.

I later used a counter to measure the conversion rate of the ADC, and found that every ~103 cycles of the 100MHz clock a new value was read in. Thus the full data set of values (left/right and up/down) was refreshed at 100MHz/(103*2) ~0.5MHz = 500kHz. This was faster than the 62.5kHz we had initially planned for, however the filter module still dealt with how much of the data to store, so that was not a problem.

4.2 Filter

The filter took in the signals from the electrodes at a frequency of 500kHz, and ran them through a bandpass filter designed in MATLAB with fir1. This set of modules was very similar to lab 5a with the filter coefficients modified to change it to a bandpass filter between 35 Hz and 1 Hz. To give the filter time for calculation, and because the analog circuitry already had a low pass filter for frequencies greater than 40 Hz at this stage the inputs to the filter were taken once every $2 * 10^5$ clock cycles, so the filter received inputs and produced outputs at a rate of 500Hz. By slowing the input frequency to the filter down I was able to continue using the 31 tap filter concept from lab5, as inputting 500 Hz and filtering 1 - 35Hz seemed more reasonable than a filter on 500kHz input filtering 1-35 Hz. If needed we could have used a filter with more than 31 coefficients, however when looking at the filtered data that did not appear to be needed. If I were to revisit this I might try keeping the input frequency at 500Hz and increasing the number of coefficients to see if that would be more effective at filtering out some of the overall drift we saw in the feature detection module.

4.3 Feature Detection

The feature detection took the two outputs of the filter - the filtered left/ right and up/down signals at 500 Hz, and stored them into two brams, each containing roughly the past 20 seconds of data. After each new pair of data points was received it then had time (running on the 100MHz clock) to average the each of the two data points over the entirety of the data stored in the bram, which it would use as the average value for the next set of calculations.

To determine the current state of the eyes thresholds were preset for looking up, down, left, right, and "tapping" the probe (which we used as a substitute for detecting blinking for reasons discussed later). For looking up/down/left and right each data point in the past ~1 second was compared to these thresholds, and if it was greater than its threshold from the its respective average in the correct direction it was considered a detection of that movement. The same thing was done for taps (called blinks in the code), however they were only counted over the past ~1/4 of a second as an attempt to allow double taps to be closer together in time than one second and still be able to be distinguished from single taps for purposes of entering the menu.

Once tallies were taken for up/down/left/right/blink these tallies were compared to each other and preset threshold to determine the state of the eyes. Blinks were accounted for first - if eyes are categorized as blinking they cannot also be looking in a direction. Next left and right alongside up and down were compared in such a way that the eyes could be for example, just left or both left and up, depending on the data collected.

This module was tested by displaying the state of the eyes encoded into the colors of the two multicolor LEDs on the Nexys4, and after some adjustments to the thresholds, the same thresholds worked fairly reliably for both myself and Crystal the ~15 or so times we stuck electrodes on our faces to work with it.

4.3.1 Dealing with Drift

One significant issue I encountered when working on the filter module was that the filtered data (and the unfiltered data) drifted gradually over time. This drift in filtered data occurred to the extent that in less than a minute the data would have drifted roughly 3 - 5 units on the 8 bit (225 unit) filtered left/right or up/down value. Given that the threshold for distinguishing left/right or up/down in comparison to forward was of similar magnitude (roughly 3-5 units) this was a significant issue. To remedy this I tried adjusting the edges of the digital bandpass filter, which did not noticeably change anything. I also tried adding a lowpass analog filter, which did appear to slow the drift a little, but ultimately I just created my thresholding method around the assumption there was drift.

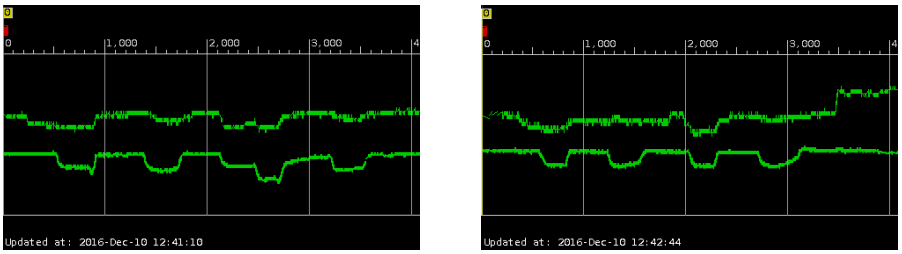
By creating a thresholding system that was based off a 20 second average of past data we ran into limitations where you couldn't just look in one direction indefinitely, as after about 10 seconds of looking in that direction, the average would be 50% looking in a direction, and would start counting it as looking

forward rather than looking in that direction. However by creating a thresholding system that was based off a 20 second average of past data I also allowed flexibility in where the user wanted to define as center - I could center looking at the nexys4, the screen or the scope, which was helpful as I was wearing the probes while debugging the code. This dynamic averaging also allowed us to not have a calibrate mode as once the probes were connected they would settle to whatever DC offset the user naturally had across their face - an important feature as the natural DC offset changed significantly user-to-user, day-to-day, or due to a slight difference in probe placement.

Ideally this dynamic averaging could be used to set up the probes and then eliminated later if the user wanted. To do this we would have to look into ways to solve the drift problem. Next steps to solving the drift problem might include adding an analog high pass filter, or expanding the digital bandpass filter to include more than 31 taps to improve the ability of the filter especially at low frequencies that cause drift. If those attempts still failed to eliminate drift the methods of detecting eye state could change to detect the edge of the DC offset and the magnitude of that jump, rather than relying on knowing what the DC offset should be for any given state.

4.3.2 Trying to Distinguish Blinking

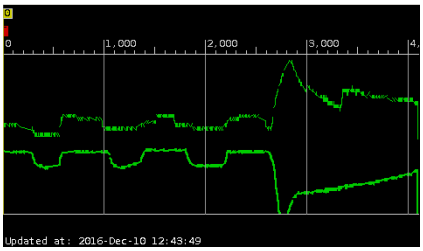
Our initial project goal set out to distinguish intentional blinks in addition to left/right/up/down. However with the probe setup we used and filtering the data from 1 - 35 Hz blinks in a normal but longer than usual matter appeared to have a similar DC offset and shape to looking up. One time when we tested the signals from the blinks appeared to be slightly more triangular than the simple DC offset of looking up, however trying this a few other times on other days when I was wearing a new set of probes, resulted in no consistent noticeable difference. Below are some screenshots of left/right and up/down filtered data as I alternate between looking up and blinking:

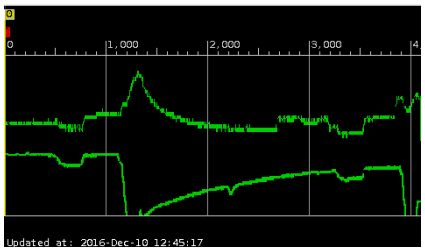


Top line: left/right differential
Bottom line: up/down differential
Sequence: center-up-center-blink-center-up-center-blink-center (once in each image)

We also tried blinking by closing our eyes as "hard" as possible (i.e. maximize

muscle contraction) these blinks were definitely noticeable, but would offset the data for a fairly long time after each blink, making the eye movements following unreadable. This more dramatic motion also sometimes resulted in a step in the average voltage read from looking forward, which was undesirable as then the data needed 10-20 seconds to re-average. Here are some examples of those sorts of "hard" blinks:





Top line: left/right differential
Bottom line: up/down differential
Sequence image 1: center-up-center-blink-center-up-center- hard blink-center
Sequence image 2: center-up-center- hard blink-center-up-center- hard blink-center

Due to these issues we decided to use tapping a probe (the user placed their finger on one of the electrodes + alligator clips for some fraction of a second) as our replacement for the blinking trigger in the code. This signal was clearer as it created a short spike of greater magnitude than looking in any direction and was reliably detectable between users, different days, and slight differences in electrode placement. It still did create some issues as the greater magnitude of the spike would offset the running 20 second average more than desired, especially if the density of clicks in the past 20 second was high.

To improve the fact that the taps offset the average I tried to adding to their detection that they should not be added into the running average, however this created issues as with user movement, the user would sometimes change their average to the extent that everything looked like blinks and then the code would get stuck in a state where it couldn't recognize any external inputs. With more time and effort this problem would be solvable (i.e. by adding a timer so that if the user had been "blinking" for too long blinks would start being average again). However, to solve the larger problem of actual blinks looking too much like looking up next steps might be looking at higher frequencies or using different probes.

5 Graphics

5.1 Overall Graphics (Crystal)

We wanted to use XVGA so we could have a larger screen to work with. This also allowed us to simply zoom the real world camera by a factor of 2 and give us some space to scroll about. To do this I used the clock wizard IP to generate a 65 MHz clock signal which all the graphics module would use and modified Lab 4's VGA module to get the appropriately timed signals.

5.2 Menu (Crystal)

5.2.1 Mechanics

The menu is a FSM that manipulates the overlaid menu sprites. It operates with an overall menu of data visualization, real world, and virtual world with submenus under the real world (camera 1, camera 2, and pictures) and the virtual world (the virtual room, maze, and a 3D maze). Depending on which direction you looked, it would prepare the next menu state as you looked in a direction and switch once you looked back center. While this worked out pretty well with button testing, it was a little harder with your actual eyes, since determining center wasn't extremely reliable. When a mode was finally chosen it changed the overall state and turned off the menu. In the main Nexys 4 module the overall state was taken in a case statement and changed the output pixel source. Since some of the graphics modules took multiple clock cycles to complete computation, registers were added accordingly.

Since there were submenus, I decided that in order for the menu to pop up reliably only when desired (and since some of the modes such as the maze or pictures would benefit from having an added clicking mechanic), it would be opened via double clicking (double-blinking at the time). Thus, I made two modules related to this ("blink" and "doubleblink", these were named before we changed to tapping the probes). Blink took the "closed" signal (when the difference was above the threshold), and compared the previous and current states, giving a single short signal when it believed the blinking/tapping had completed. Double blink was a very simple FSM that changed to a one-blink state when a blink happened and if another blink happened within a specified time frame the double blink single was given, else it returned to the no blink state.

5.2.2 Reading Sprites

Since we knew that the camera code that Weston provided worked only with a frame buffer placed in the BRAM which would take up ~70% of the space. Thus, I run-length encoded the menu sprites through a python script, generating .COE files for a ROM to read. With a few more commands I determined the most efficient number of bits to have per run-length would be 8 bits max. Originally since there were going to be only two colors I was trying to use a run length

encoding that switched between the two colors (no color bit basically), where when a run of one color had a length longer than 255 pixels there would be an entry after that was 0 (to signify the other color had a run length of 0) and then the next entry would continue the count of the first color. I saved these all using a ROM IP. The plan was to have, as a countdown occurred starting at the previous run length, the next entry would be read to prepare since it takes several cycles to read from memory. However, since I had run-lengths of 0, I either had to read several entries ahead or to find a different method.

In the end I added a bit for the color, which eliminated entries of run length 0 but also increased the bit width of each entry to 9. There still were issues though, and eventually figured out it was when the run lengths were either 1 or 2. It probably was still the same issue as above, but instead I just went through and hand-corrected entries. If it was 1 and the previous entry was the same color, I decreased the previous entry and increased the next; if the previous entry was a different color, I removed the color entirely and increased the count of one of the surrounding entries. Similarly, for entries of 2 with the same previous color I altered them accordingly and with different previous colors I went through the pixels and decided whether they'd be better/smoother having no color change there or if they should be expanded into three pixels, and if so, which direction to expand them in. This process was aided by a python-generated .txt file with 1's and 0's based on the run lengths (since I started originally with a picture seeing distinct pixels as 1's and 0's helped make it obvious what was going on at that scale). All these scripts and original files can be found in the "python" folder.

5.3 Data Visualization (Crystal)

For the cartoon eyes I made an eye module that makes a single eye and then duplicate it for the pair of eyes. The calculation was done in two cycles. First, it determined how far a pixel was from the center of the eye whites and, depending on the eyestate, how far it was from the center of the eye pupils. The pupil locations were determined parametrically/through parameters depending on where the center of the eyes were placed. In a second cycle it would determine whether or not the pixel was within the specified eye radius. This module was implemented with relatively no issues.

5.4 Real World (Elizabeth)

The real world takes as inputs the eye state, the camera data, and the 65 MHz XVGA clock. It was modified from Weston's camera code found here: https://www.dropbox.com/sh/lndgbb8hylncgp6/AAD2tzv_HgQlrHxNSZA0N_BQa?dl=0 The unexpected difficulty I ran into with the camera was actually primarily having links to incomplete and out of date files, but once I acquired code that had the top level module written the camera was significantly easier to understand.

The first main modification I made was "zooming in" on the camera by a factor of 2. To do this I divided the memory location being sent to the bram to request the image for the coordinate of (x/2, y/2) rather than the coordinate (x,y). This meant that each pixel of the camera was displayed as 4 pixels on the screen, allowing the camera data which only fills up a VGA screen of 640x480bits to fill up the entire XVGA screen of 1024x784bits and leave a little extra not displayed.

The second main modification I made was allowing the camera to pan around this enlarged image. To do this I again edited the location of memory the display module was retrieving from by adding counters for when the left/right/up/down signals were true, and then adding those counters to the variable that controlled what memory location was pulled for each pixel. I also put limits on these variables to not allow the user to pan outside of the shot of the camera. Through testing I found a speed that seemed reasonable for the user to pan up/down and left/right.

5.4.1 Two Cameras

To alternate between displaying two cameras I created duplicate modules for each camera up to the point the camera data was stored in the bram. (This means I duplicated the camera_configure and camera_read modules.) Then before the camera_address_gen where the bram address was generated I used the state of the menu to decide which camera's data to pass to the camera_address_gen module.

The other complication of having two cameras was that they both had to run off the same camera_clk_out, as the JB[7] pin was the only pin on the nexys4 that could handle the required data transfer. However, it worked out that both cameras were similar enough that using the $\mathrm{camera}_{c}lk_{o}utofcamera1forcamera2workedwithoutanyapparentglitche$

Near the end when we copied my code over into Crystal's project we thought we had another camera issue as the camera was displaying only a black and very dark blue image, however I discovered (using 'default_nettype none) that that was simply because we had failed to declare the wire that transferred the image from my module to the menu, and thus Verilog assumed it was a 1 bit wire.

5.4.2 Servos on Cameras

To allow the user to have a wider range of view in the real world I sewed the cameras to the horns of finite rotation servos. I used one HiTec HS-311, and one Vigor VS-2A. The two servos functioned similarly though they had slightly different max and min PWM durations, and the HiTec HS-311 functioned smoother as it had a higher torque (it turned out the camera wire produced some torque) and was a newer servo.

I created and controlled the servo signal from the video_playback module, as once the camera reached a left/right limit within the current camera position I would check the servo position and if it was not at a limit (of the possible PWM signals the servo could handle) modify the PWM signal to rotate the

motor slightly in the direction of the user's desired movement direction. When troubleshooting the PWM signal I first ran into the issue that the servo drew more current than the batteries could provide at the voltage required, so I switched to powering the servos from a 5V external voltage generator. I thought I had checked the spec sheet that this voltage generator was connected to a non-floating ground, but when the ground voltage from the ADC produced odd oscillations with respect to the ground of the power supply, I realized that was my issue rather than the code generating the servo signal. This had me stuck for longer than I should have been as a kept trying to troubleshoot my servo signal rather than the grounding of all my power supplies.

5.5 Virtual World (Crystal)

Unfortunately, the virtual world modes weren't successfully implemented. Here we look at the ideas and math behind the process.

5.5.1 Basics

To start, the viewer has a theta and phi angle associated with the direction they are currently looking at. These two values come from the standard spherical coordinate representation of them, where theta is the angle from the positive x-axis to the vector projected to the xy plane and phi is the angle from the positive z-axis to the vector itself. After consulting with a friend as to what would feel most natural, they believed that turning "left" or "right" would simply change the angle theta while "up" and "down" would change phi (rather than, in contrast, having wherever you were looking have left & right be relative to the current screen), as though there were a camera on an up-down servo and the combined component be mounted on another left-right servo.

There were various ways to then determine whether a pixel was in a rectangle or not, which, in retrospect, some might have been more implementable, though with cons. One method would be to project the corners of walls onto the viewing screen, and then one of the methods to create a line between neighboring corners. I didn't originally choose this method mainly because it seemed hard to fill in the wall faces/determine whether a pixel was inside a polygon or not without complicated math. Since I was hoping to implement a maze there would be no point if they could see through the walls. I also had originally thought that since the planes walls were in would have a normal vector "pointing towards the viewer", thus able to describe the planes in spherical coordinates (how far away the plane is and then in which direction) which I hoped would be easier as the viewer was in spherical coordinates, and somehow transform a point projected onto them into one relative to the plane they were on, then easily determine whether this point was in a rectangle. This however, I soon realized, wasn't as straightforward and involved way too many conversions between spherical and rectangular coordinates, and also would make it a little difficult for defining the walls on the plane when changing the location of the viewer (relevant for the maze).

I finally decided on a scheme that went like this: depending on where the viewer was currently looking, a vector from the viewer to a pixel on the screen (so depending on hcount and vcount) would be created in a vector module. Then the vector would be projected to a plane (defined by a normal vector and a point on the plane (in rectangular coordinates). Finally, a module would determine whether or not this projected point was in the rectangle or not, and color in the original pixel accordingly.

5.5.2 Math

For the first module, vector, which would be used for all walls, I used Rodrigues' Rotation formula. To rotate a vector $\mathbf{v}$ by an angle *theta* around an axis unit vector $\mathbf{k}$ (determined by the cross product of the original vector and the desired vector), we have

$$\mathbf{v_{rot}} = \mathbf{v}\cos\theta + (\mathbf{k} \times \mathbf{v})\sin\theta + \mathbf{k}(\mathbf{k} \cdot \mathbf{v})(1 - \cos\theta)$$

To determine the vector which the present pixel is part of, I started with a vector from the center to the point (512, 512-hcount, 384-vcount), which is basically the screen centered around the vector pointing along the x-axis, and then first rotated it by the viewer's current theta around the z-axis, and then rotated the resulting vector by phi along the axis $(\sin\theta, -\cos\theta, 0)$, which is the vector perpendicular to both the z-axis and the current vector (in the direction of their cross product). The result of this module would be a vector relative to the viewer's location pointing in the direction a pixel was aimed towards. While this took several cycles to compute, since theta and phi changed at a much slower rate (~half a second of cycles before changing values), I didn't really bother with registers for this calculation, since it would quickly settle down.

A plane is defined by the equation $(\mathbf{p} - \mathbf{p_0}) \cdot \mathbf{n} = 0$, where $\mathbf{n}$ is the vector normal to the plane and $\mathbf{p_0}$ is a point on the plane. In other words, $\mathbf{p}$ is on the plane if the vector from $\mathbf{p}$ to $\mathbf{p_0}$ is perpendicular to the normal vector. We can express the point $\mathbf{p}$ as $t\mathbf{v_{rot}} + \mathbf{c}$, where $\mathbf{c}$ is where the viewer is located in the world. Solving for t we get

$$t = \frac{\mathbf{p_0} \cdot \mathbf{n} - \mathbf{c} \cdot \mathbf{n}}{\mathbf{v_{rot}} \cdot \mathbf{n}}$$

After calculating t we can multiply it by $\mathbf{v_{rot}}$ and add it to $\mathbf{c}$ to get the actual point $\mathbf{p}$ that is on the plane.

Finally, to check whether a point is within a rectangle we can do the following calculations. If the point p we are checking is coplanar then if given points p0, p1, and p2 where p0 is the corner between p1 and p2, then if we make the vectors $\mathbf{u} = p0 - p1$ and $\mathbf{v} = p0 - p2$, p is in the rectangle iff $\mathbf{u} \cdot \mathbf{p1} \leq \mathbf{u} \cdot \mathbf{p} \leq \mathbf{u} \cdot \mathbf{p0}$ and $\mathbf{v} \cdot \mathbf{p2} \leq \mathbf{v} \cdot \mathbf{p} \leq \mathbf{v} \cdot \mathbf{p0}$. We do this straightforward check to determine if the projected point from the `project_to_plane` module is within the defined rectangle.

5.5.3 Implementation Attempts and Problems

When I did this the first time, I imagined the screen was placed one "unit" away from the viewer, and thus the original vector needed to be scaled down by 512 for the "actual" length. Since this would result in values less than 1, I pretended the decimal point was simply 9 bits from the end. For sine and cosine, I again made a python script, and finding that $\pi/2$ in binary was close to 202 in binary with 10 decimal points, to make a corresponding COE file for angles from $0 * \pi/202$ to $807 * \pi/202$. Thus, in general, angles were expressed in terms of multiples of $\pi/202$.

At any rate, the vector module was wrought with issues, first mainly because signed arithmetic was difficult with verilog. One issue was that even though I multiplied signed values and then right-sign-shifted (to account for the extra decimal points), and thus needed a smaller bitwidth, if I assigned this value to a smaller bitwidth than the sum of the factors since I no longer needed such a large bitwidth, it would no longer be correct, since the sign bit was the most significant bit. Also, an issue that came up more later, was that decimals were getting hard to keep track of.

Wanting to stop using signed values (although it's inevitable for vector), I had originally decided to keep the possible positions only in positive coordinates, and only go up to 64 (6 bits) in any coordinate. It was in `project_to_plane` that I would try to scale the extra 512 factor. I thought of it as more of projecting down to the smaller space from a larger screen 512 away, since the vector would be the same whether it was at 512 away or 1 away. Since it was unlikely that after projecting to the plane the points would be on lattice points, I had added two more bits for decimals. Again, keeping track of how many fractional parts was tedious and a source of errors. There was a bigger issue though, but I didn't learn this until later. There also was a clash between the signed and unsigned wires, where verilog could not interpret math correctly between a mix of the two (even if they all end up positive, as I was trying to make happen, it just didn't work). I didn't realize the severity of this issue until I was debugging in_rect, which is an extremely straightforward module but was working really strangely, due to this signed/unsigned clash.

My second clean attempt had me restart and give up on only having positive values after the vector step. I also decided that instead of thinking of decimal points I would think of it as scaling up by 512, so now I would have 16 bits to describe a point instead of 8 bits, and that everything would be signed. Now I would not need to keep track of decimals as much (I would still have the issue with vector of signed multiplication and then division being inaccurate if you cut off the number of bits).

I soon realized a big problem was actually the division required for the `project_to_plane` module. There are still a couple things I would like to try if I have more time in regards to making this module work, but what I did do to start was involve a division IP. Since it seems like I'll have to use one for each plane and since division is extremely expensive, if I want to use many many planes I'll probably have to look into adding a frame buffer (which may take up

too much space if the camera is also using one, though maybe we can use the same space since only one is displayed at a time) and changing between which plane I'm using the division IP on at any point, and maybe switch really quickly between showing each plane.

I used the radix-2 implementation with fractional parts, but this gave me some issues. It seemed like since it was fully pipelined, one quotient would come out each clock cycle, for a division that started ~20+ cycles before/latency. I thus made a memory so that it would cycle through and store the coordinates so that t could be multiplied by the correct number. However, I was getting a problem wherein the diagonal lines were at the wrong slope but occasionally dropped to the right value... At first I thought it was because the division wasn't getting enough precision, but after using an ILA I determined that the latency wasn't the same for all divisions! Or that at least the division only occasionally matched up. I think I either need to read the specifications of the IP again to see why this is happening or perhaps switch to trying out the High Radix implementation instead. I also made the modules less general and had more hardcoded, such as only creating planes that are parallel to either the xy, yz, or xz planes, hoping this would reduce the range of possible issues.

5.5.4 Maze Mechanics

Even though the basic wall generation was not quite done, the maze mechanics have been thought out. Basically depending on the angle you're looking at you travel in different directions with a click. With maze3d you can also possibly go up/down. Each cell is represented by a 6 bit number, with 0 corresponding to an open path in that direction from the cell and a 1 if the path has a wall in between. Unfortunately, without solving the problem of division, it isn't likely that very much of the map can be generated. With this in consideration, it's possible to instead just generate the walls of nearby cells, as often further ones are simply un-seeable, blocked by the ones in front. This would require an additional model to convert an entry in the bram to a form that the modules would understand. Probably project to plane won't change even as you move around since the cells would be at the same distance away, but `in_rect` would depending on the state of the walls. It might even be that we would not need **c** if we can calculate everything relative to the viewer.

To figure out which wall is in front, we use the t scaling value that was saved. Since at any one pixel the base vector is the same length, comparing t in a single clock cycle will be sufficient to find the wall that is closest (that is, if the walls are different colors). If the walls are the same color then as long as there's one wall at that pixel it can be colored in; it doesn't need to know which wall it's on. An important note about t is that it can be negative, which means there's a wall behind you at the same angle. We therefore disregard any pixels from `in_rect` with negative t from the corresponding plane.

6 Lessons Learned and Advice for Future Projects

6.1 Elizabeth

Through this project I reinforced and grew my past knowledge about amplifiers, filters, and PWM control in addition to gaining experience with dividing up and defining large software problems, understanding Verilog functionality and features, and troubleshooting when things didn't go as I expected. I think this is the largest code-based project I have ever created which seemed intimidating at first when I had to define every single module, it's memory and how it would interface with other modules. However I found that by stepping very slowly through each module I could gain a good understanding of it.

In specific, through creating IPs for clocking, the xadc, many ilas, and moving the camera IPs from one project to another I got more comfortable with reading online specifications for timing and troubleshooting hints. By working with the camera code Weston wrote I got practice going through code to figure out which parts I needed to modify in which manner - I made flow charts with inputs and outputs and had to understand what each one did in order to add a new camera. I also debugged using numerical output on the screen of the nexys4, the ila, and the logic analyzer. By using all of these methods I learned more about the strengths and weakness of each and feel like I have a better understanding of how to choose one in the future.

I also learned from how this project helped us manage our time from the abstract to the proposal to the checklist, as I've previously struggled to clearly divide up and define timeline for large projects and have seen others do the same. I thought the act of writing checklist was an interesting checkpoint in the middle and may even try that with the high school physics students I'm teaching!

I think this project could be improved through further investigation into the difference between sensing blinks versus looking up. One could try different probe placement, different probes, or different cutoff frequencies for the filter as it would be nice to detect blinking (possibly detecting both intentional and automatic blinking to). If the signal was better, possibly through other probes, or more amplification and filtering before routing into the Nexys4 I think the project could be extended so that we could not only group a look into the boxes of left/right/up/down but define its magnitude and direction more precisely which would allow for smoother manipulation of something like a computer mouse. I think the camera "real world" side could be made more interesting by attaching the camera to a drive base or two a second servo, allowing a translation form of motion or a second degree of rotation.

6.2 Crystal

I definitely underestimated the difficulties of generating 3D graphics would be – a lot of things that might be relatively simple with a computer are less trivial on an FPGA. There are a lot of considerations, from space issues (making

sure everything fits in memory) to computation limits (how many divisions one can do, for example), and timing (time to read from memory or to process a computation). I'll probably look into what other people do for these sorts of problems instead of figuring out all the math on my own, which, though probably correct, are likely not to solve the problem in the most efficient and effective way possible.

Scaling is also a problem I hadn't really thought about – in my mind after making a single wall it would be extremely easy to to just add more modules of the same type, but it's really not that simple. Whether it takes up more resources (I originally had thought to read from the sprites all at the same time on one memory to save on space, but you can't actually read that many entries at once, even if it's a read-only ROM), and making more just makes copies and you don't actually save space, or whether it takes up computation power (if I really want to do the more complicated graphics, having a divisor module for each wall would definitely not be practical).

I wish I had been taught how to use the ILA and IPs in general earlier/during the courses, as I hadn't learned about the ILA module until relatively late, and it was only then that my debugging turnaround sped up.

If I were to do it again, I would probably have liked to change the goals a bit, making a single plane move in perspective as the goal with multiple walls as a stretch (and a maze as a superstretch).

Like many others I'm sure, I'd advise others to start early. Maybe not necessarily be on top of things always (though it's better to be) but to definitely have a good understanding of the difficulty of the problems each section will face so making a timeline will be more effective and accurate, and so you're less surprised last minute when things aren't working out so well. Maybe also take lots of pictures/document your work throughout, as then when you're writing your paper without an FPGA handy you have pictures to insert (but also it's just nice to have records). Oh another annoying thing I had was that my athena account had reached its quota in terms of memory and so Vivado would get stuck or crash at certain tasks a lot, so make sure you're not running into this problem (I doubt it'd be very common problem but it was really frustrating until I figured it out).

7 Code

You can find our code here: https://www.dropbox.com/sh/3yklng80mfg4l1r/AADpYEYmVGUQvc-41b-6CVNFa?dl=0.