ARTable Final Report

18-500 Group 3B
Steven Fu, Steffen Holm, Evan Li

Embedded Device

The embedded processor that we chose to use to process input from the camera and output frames to the projector was an Odroid XU4. After doing research and comparing to other embedded platforms, such as those from Raspberry, Beaglebone, Odroid, and ASUS, we found that some of these other platforms have less community support and examples of past success for similar parts of our project, and that some projects have used versions of the Odroid platform to different degrees of success. Because we know that some past projects are able to process Kinect video streams at a reasonable rate given an older version of the Raspberry Pi that has a slower processor, we can estimate that the latest version (Raspberry Pi 3) would be able to process the stream more quickly, as well as output frames to the projector at a reasonable rate for usage. Some issues that other platforms have run into are a lack of driver support or unstable drivers (Beaglebone), lack of performance (Raspberry), and lack of updated documentation and software (ASUS).

While the Odroid shines in all of these aspects, it unfortunately has USB port instability issues; some users have reported that USB ports will work only upon cold boot, or only if multiple devices are plugged into the USB ports instead of a single one at at time. Because we wanted to choose a relatively stable and mature platform that has some community support and updated documentation, the Odroid XU4 is the appropriate choice for us. Furthermore, it is faster than these other processors for processing the video stream, so these advantages will allow us to save valuable development and debugging time. Although it is not as powerful as other platforms out there like the Nvidia TK1, we believe that it will still be able to take input and output frames at an acceptable rate such that it does not impede on the user experience. Another factor that we considered was cost, and the Nvidia TK1, although a very powerful board that would also allow for CUDA support to accelerate computer vision algorithms, is prohibitively expensive at almost three times the cost of the Odroid, which is already two times the price of the Raspberry Pi.

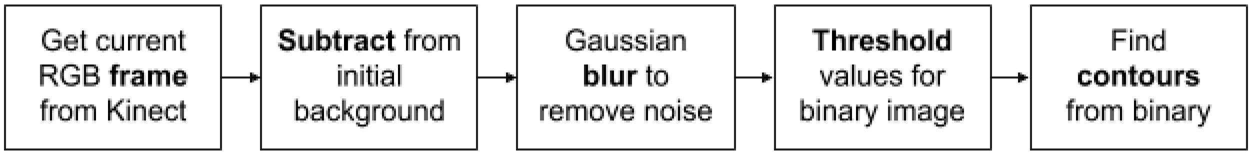
The Odroid ended up being a good choice for us because it was able to simultaneously process the input stream from the Kinect and output frames from the demo applications through the projector via HDMI. Throughout the development process, we ran into several issues when using the Odroid. When first setting it up with the necessary drivers and libraries to interface with the Kinect, we discovered that there were several packages with version-specific dependencies that had to be installed in a particular order. To complicate this, the Odroid did not support OpenGL and specific build flags needed to be set for several packages to use OpenCL instead. Even with this, installing the dependencies for libfreenect2, a set of open source drivers for the Kinect, did not allow it to function and connect to the Odroid. This took significant debugging time, reinstalling a clean copy of the operating system multiple times, and testing of different versions of dependencies until the board was able to detect the Kinect device when plugged in. After this, the next challenge was getting the packaged demo executable to run on the Odroid through OpenCV so that we could use some of the computer vision functionalities. This required modifying the demo executable and interfacing that through OpenCV's frame interface, while ensuring that both libraries used OpenCL rather than OpenGL.

After these initial hurdles were navigated, the rest of the development process on the Odroid was relatively simple, and interfacing with the projector was simple because there were HDMI drivers that worked well. The other challenge that we had to work around was the unstable USB ports. Although we were initially skeptical that this could be the case for an embedded device that had worked so well so far, when we powered down the device after development we realized that the issues were real and ended up having a small impact on our development time. We found that when the Odroid was powered off and restarted, the Kinect would sometimes fail to be detected by the system. This was fixed by a specific process of powering off the board, disconnecting the Kinect, ensuring that some device (in our case, a keyboard) was plugged into the other USB 3.0 port, plugging in the Kinect, and then powering on the board again. We did not attempt to debug or find a less complicated process of solving this issue because it was a relatively quick process despite being an annoyance, and also was not reproducible one hundred percent of the time.

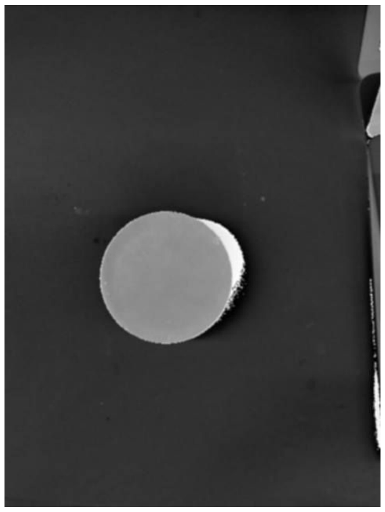
Camera System and Computer Vision Algorithm

For the camera, we used a Microsoft Kinect Sensor v2 for Windows. Initially, we chose this because it would allow us to get both a depth and RGB image from the camera, so that we could not only detect objects in our beginning stages when they are a uniform color that is different from the table, but also detect objects even if they are the same color as the table or projected images. We considered using only an RGB image, but this would allow us to easily differentiate between the projected images and placed objects if they were to be of the same color, or if the table were a nonuniform color to begin with. This was connected to the Odroid via USB 3.0 to maximize bandwidth utilization, as the Kinect captures RGB frames in 1920x1080 resolution at 30 frames per second. Thankfully, we were able to use a set of open source drivers for Linux to interface with the Kinect, since Microsoft only provides Windows drivers for the Kinect.

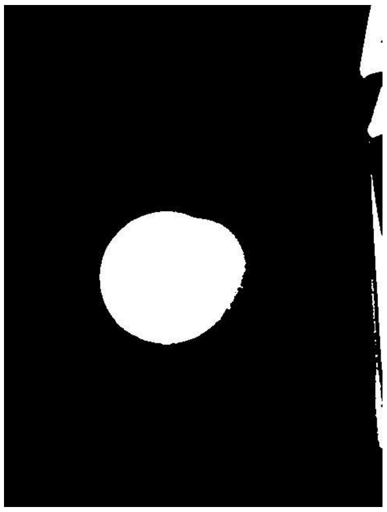
We decided to develop an initial object detection algorithm that only used the RGB stream as our base for detecting objects. Our initial algorithm used background subtraction to find the difference between the background and the current frame, and find the contours in the differences. The process from the current frame from RGB feed to bounding boxes/contours is as follows:


Get current RGB **frame** from Kinect
Subtract from initial background
Gaussian **blur** to remove noise
Threshold values for binary image
Find **contours** from binary

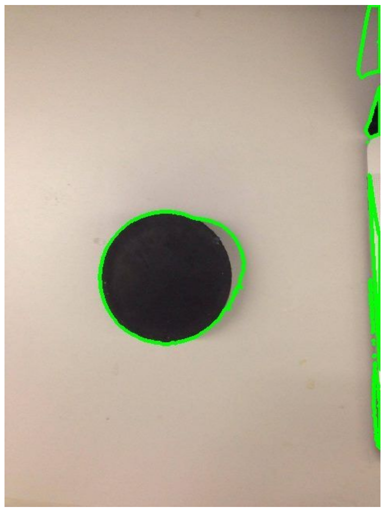
Background Subtraction



Otsu’s Thresholding



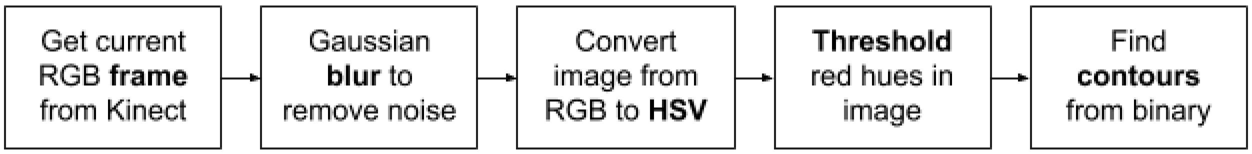
Finding contours

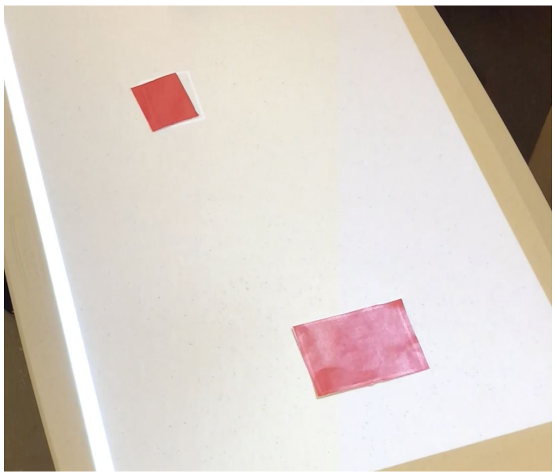


The one flaw with our algorithm was that it did not account for projected images and only assumed user-placed objects would appear against the background. If the game was run simultaneously with this algorithm it would detect the components of the game as obstacles. This meant the spaceships and bullets could have different properties from what we wanted, which could prove problematic. After finishing this initial algorithm and understanding its limits, we aimed our sights towards the depth stream to see if we could detect objects using a similar background subtraction algorithm.

Using the depth stream instead of the RGB stream meant that we wouldn’t have to worry about the image projections interfering with the object detection. The depth stream would also provide an extra dimension (height) which could possibly lead to other functionalities in the game. There was even a registration module within the libfreenect2 drivers that would essentially combine the RGB and depth streams to form one combined image which give us lots of freedom and options with our algorithm. However, after implementing our algorithm with the depth stream, we discovered that it was not suitable for our purposes. Because we were mounting the Kinect close to the table, the depth stream was unusable. The Kinect is designed for large room usage, and for distances of approximately one and a half feet or less, there was no available depth discernment from the camera. Additionally, there was a large amount of noise, since the depth stream came in at a resolution of 512x424, so it was very difficult to use in conjunction with the RGB stream; there was also not enough granularity between different depths, so objects placed on the table that users interacted with would have to be very tall to be detected by the depth camera. As a result, we chose to ditch the use of the Kinect depth stream, and improve our algorithm with the RGB stream instead.

Our goal now was to differentiate between the objects that were placed and those that were projected. We opted to detect only a particular color - red, and avoided using red in the demo application. This decision turned out to solve our problems: placed objects were easily detected, and the algorithm could ignore projected objects. The only requirement we had for this algorithm to work was that the projection could not contain any red. Our new algorithm that converted a frame from the RGB stream into bounding boxes is as follows:

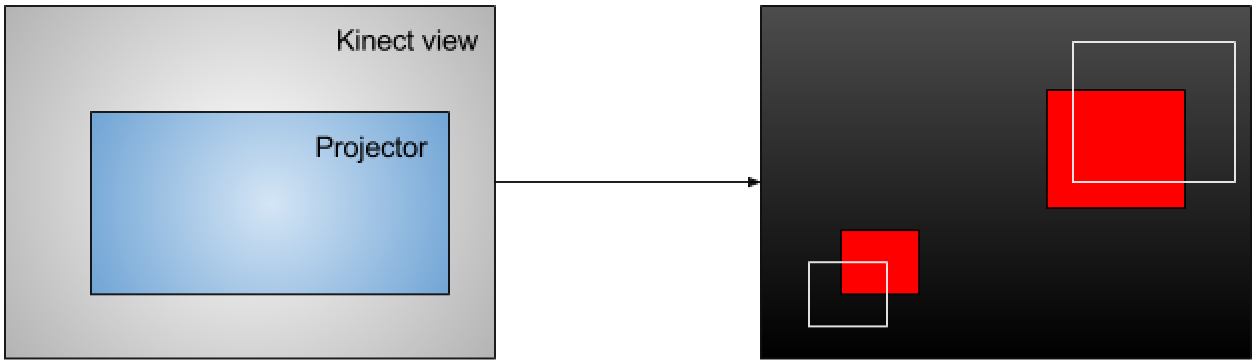

Get current RGB **frame** from Kinect
Gaussian **blur** to remove noise
Convert image from RGB to **HSV**
Threshold red hues in image
Find **contours** from binary



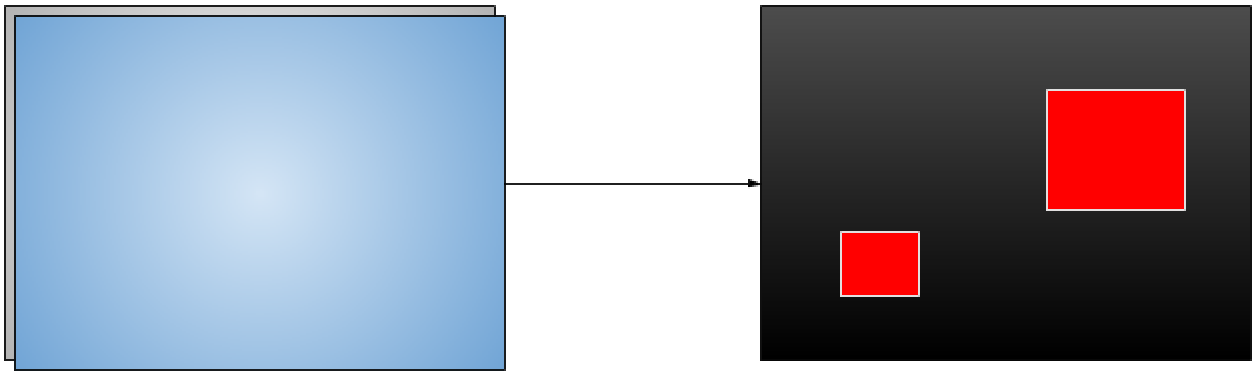
The conversion from the RGB color stream to the HSV color stream was essential for finding the right ranges of red to detect. The HSV color model more accurately describes human-perception of color and the values that we threshold at in the HSV color space provide a better range than the RGB color space could. From this algorithm we were able to obtain a very accurate bounding box of red objects that we placed as well as ignore objects that were projected on the table. The only issue we had with the algorithm was that we noticed that hands were detected whenever they came into view. For instance, while picking up a red object and placing it in view, the hand would momentarily be recognized as an object as well. We could have improved our algorithm by making the range of recognized red more strict.

Another problem that we ran into after setting up our entire system together was that the kinect's view of the table did not perfectly align with the projection. This was solved by providing the user with an initial calibration step, where the user could input the bounding corners of the projected image. From those 4 points a projective warp matrix could be calculated which would be used to warp each following image from the kinect stream.

Offset Kinect and Projector views leads to offset bounding boxes on objects



Calibrated Kinect and Projector views results in accurate bounding boxes on objects



Projector

For our projector, we used a BenQ MW820ST that was borrowed from the IDeATe department at CMU. This was sufficient for our purposes as it provided a bright enough projection so that users would be able to easily see the projected image, and also provided a large enough image to create an immersive experience at the short range that we were projecting from. Another key feature was keystone correction, which allowed us to mount the projector close to the stable stand of our wooden mount, while projecting an undistorted image at an angle to the surface of the table. Towards the end of the semester as we were integrating the different parts of the project, we ran into an issue that we were able to work around, but had not foreseen. Because we had originally planned to use the depth stream to correctly differentiate objects placed on the table from images projected from any demo application, this would not have been an issue.

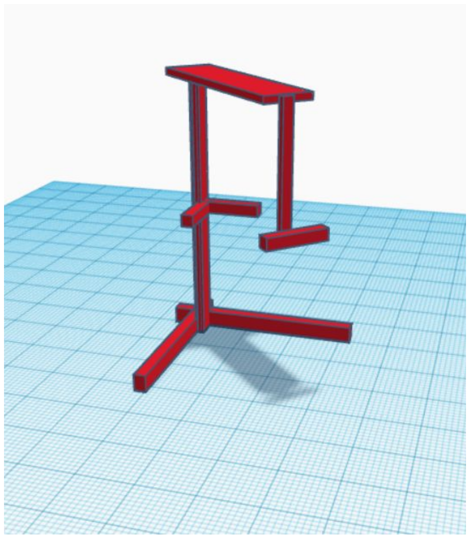
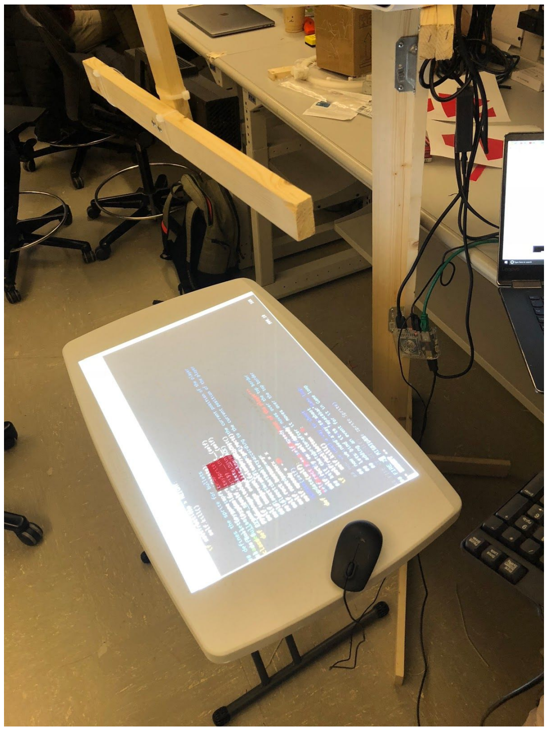
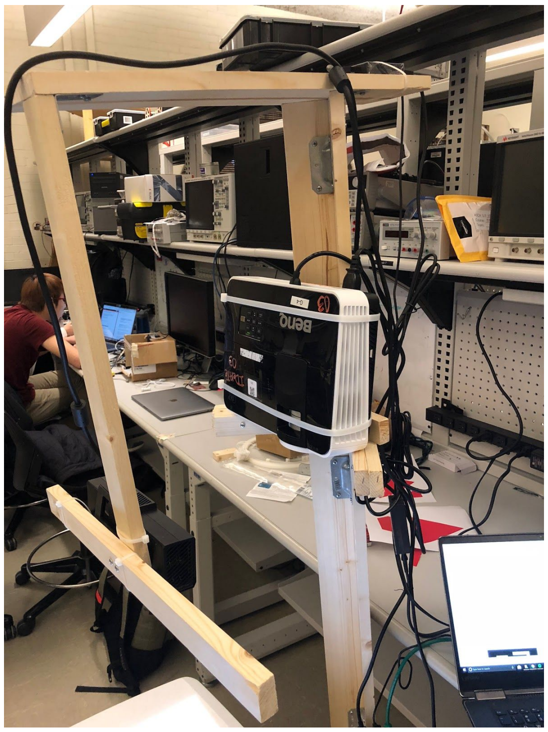
However, after testing that and pivoting to an alternate approach in which we used only the RGB stream from the Kinect, we realized that there were difficulties in ensuring that the computer vision algorithm was robust enough to detect red objects. In that development process, we quickly found that there was shifting of the detected object in the vertical axis of the camera view between, even though each component of the system was stable and not moving. This was initially difficult to debug because even when viewing the RGB stream from the Kinect of the projected image, we attributed the different colored artifacts in the stream to feedback from viewing the detected image through the projector. Although this was true, the real cause of this was not discovered until we attempted to photograph and record the projected image through a phone camera by chance.

We found that there were scrolling rainbow lines, not only in the preview of the photo or video, but also in the captured photos and videos in our phone. After doing some research, we came across people who shared their experiences and failures online of attempting to film a projection, and a few experts who said that this was normal due to the scan line rate of the projector not matching the shutter speed and frame rate of the camera. However, because none of us had done research into how projectors and cameras physically worked, we did not know how to work around this issue or why it would occur, and did not know if it was solvable this late into the semester. Fortunately, we owned a phone that had a camera that could record in different frame rates of 24, 30, 60, 120, and 240 frames per second and we experimented to see if we could figure out how to solve the issue, while also researching to learn how projectors worked. We realized that this was not an issue that, given our resources and time constraint, was fixable.

The Kinect records at 30 frames per second, but we found that recording a projection of a mostly black screen through a phone camera at 120 frames per second mostly mitigated the rainbow effect. This is a combination of how our specific projector and camera worked. Our projector was a single chip DLP projector (as opposed to a three chip DLP, LCD, LCoS, or LED projector), and thus had a spinning color wheel - this was what we were seeing through the Kinect and our phone cameras. The scan line rate of the projector was listed by BenQ to vary between 23 and 120Hz, and together this information allowed us to figure out that when projecting a mostly black image or screen, the scan line rate was at 120Hz; recording this at 120 frames per second would then sync up the camera and the projector, making the rainbow effect almost disappear. This was a very specific set of constraints that would not be possible to maintain through our demo application, and because we were using a Kinect that recorded only at 30 frames per second, we would not be able to fix this issue. However, we found that our computer vision algorithm was robust enough that even though occasionally there were slight shifts in the detected object location, they quickly reverted back to the correct location; furthermore, because the demo application took in the detected objects as input slower than they were actually detected, sometimes those shifts were missed in the demo application and the location did not change between frames.

Physical Mount

For the physical system mount that would hold our entire system, including the Kinect and projector, we purchased different sizes of wood, and with the advice of a wonderful employee named “Chuck” at Home Depot, constructed the structure using wood, angle brackets, zip ties, and screws. The design of the mount presented particular challenge as none of us had prototyped a physical structure like this and realized that building circuits was much different than building a mount. We realized that this was a valuable experience for all of us that we would not have had if we didn’t create an end-to-end system. Another challenge we had to navigate but quickly found a solution for was the splitting of wood when screws were inserted, which we fixed by drilling holes for the screws to go into first, as advised by a fellow classmate. The final build can be seen in it’s 3D design and final implementation below.

Table

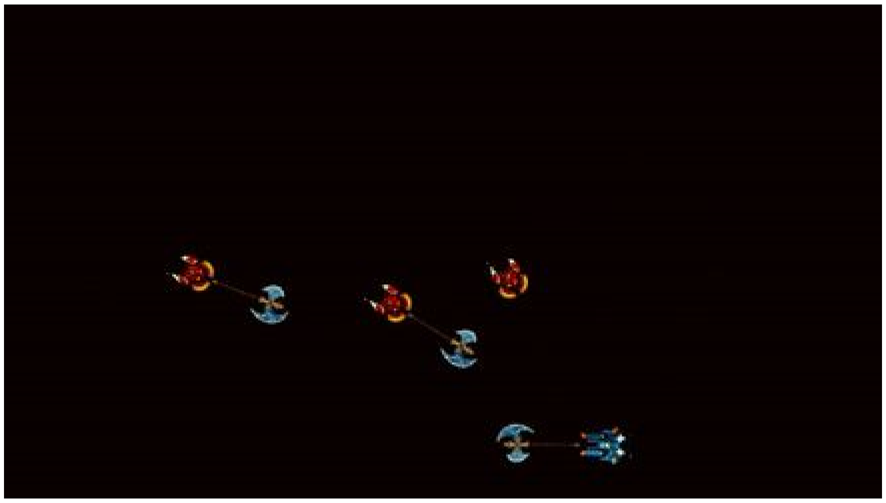
We used a table that was 20x30 inches, which was determined to be the right size for our setup. This is because we found that a reasonably sized demo application could be projected at a distance of about two feet, with the Kinect also approximately that distance from the surface of the table. This would result in a projected image that was that size and approximately that aspect ratio, with only a bit of the top and bottom of the table not being covered by the projection. However, once we developed the full game demo, we realized that the players needed to have a stable surface for the keyboard; the original table, being so small, would not have been conducive to a good user experience, and on demo day, we used a much larger table from the ECE department. This presentation made the table projection less immersive but resulted in a much more playable experience.

Inter-process Communication Between Algorithm and Application

To allow communication between the computer vision algorithm and demo application so that data (detected object locations) could be sent as output from the algorithm into the application as input, we had to figure out some way of performing inter-process communication that was efficient and portable across platforms and languages. We ended up choosing between sockets and named pipes, settling on named pipes (FIFO) because they were easier to set up and manage between the code running the algorithm in C++ and the Python/Pygame process running the demo application. An additional benefit was that we did not need guarantees on whether or not the data was written, since the C++ code would be writing data faster than the Python game could read, so it would acceptable to skip data in favor of writing new data from the next input frame. We also had to ensure that neither process prevented the other from either reading or writing, so that the algorithm could continue outputting data while the game read data.

Demo Application

The demo application was developed in parallel with the rest of the system in order to adapt the application to the strengths and weaknesses of the pipeline as it was being developed. The demo needed to be simple enough that users could quickly understand how our system was working, but interactive enough that users could experience the demo multiple times and get something new each time. In light of these requirements, we chose to develop a two player game that ported the real world objects into virtual game objects that modified gameplay. The type of game to be developed was a 2D top-down shooter where each player controlled a spaceship that was trying shoot down the other space ship. The gameplay became dynamic once the real world objects were ported over to become virtual obstacles that would block bullets and cause the user to die if they collided with the object. The first prototype of the game was developed in Unity - a 3D game development platform - and can be seen below with simulated enemies.



This version worked on the computer it was developed on, but when porting the game over to the ODROID we came to the realization that Unity does not support ARM. So this version of the game had to be abandoned, battle axes and all.

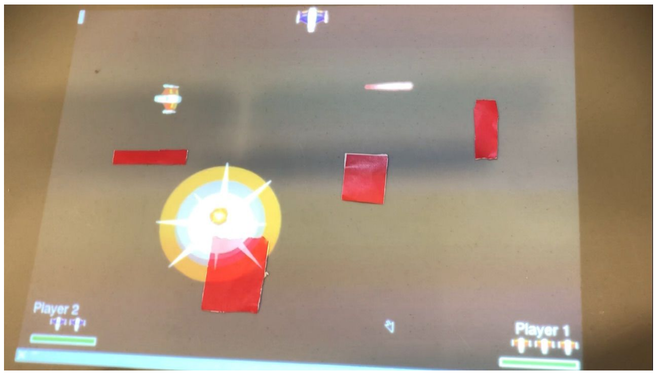
The next version of the game was designed in pygame and because we had lost a lot of time with the previous version of the game, we decided to take an existing game and mod it for our demo. The initial game was a one player game where a player moved left or right and shot asteroids. The first step to developing the demo was to understand how this game was structured and plan out how the restructuring would look like. The main game loop processed user key input, updates projected movement, detects collisions, updates each object's location, and kept track of each player's lives and health, and redrew the game frame with all sprites at a constant FPS.

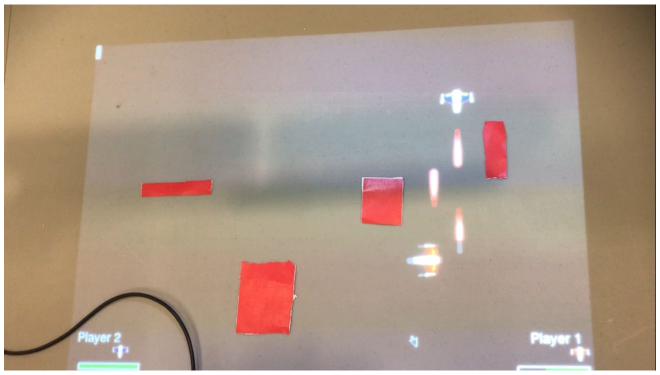
One of the key changes was the "Obstacle" object that was going to represent our detected real world obstacles. It was created as a modified "Mob" object without rotation or movement that spawned where the named pipe specified. The collision detection was originally handled using a sprite collision detection function that compared the sprite .png masks and determined collision. This needed to be replaced with a box collider whose width was set by the named pipe input. In order to work on the game while the computer vision component was being developed, we created an bounding box server that wrote simulated bounding boxes to a named pipe.

Expanding movement from the X to the XY plane was fairly straightforward, but implementing responsive took a little more care. The ship now needed to keep track of its orientation and transmit that information to the bullets it created when shooting, so that they could be rotated and fired in the correct way and not register collisions with the ship that fired it. Once this was set up, the game was expanded to support two players - duplicating the health bar and lives icons and making aesthetic revisions to present a clean simple game. One design choice was the decision to flip the orientation of the game when it became clear that only one keyboard could be supported so both players would need to be next to each other and this was done most comfortably on the opposite side of the mount.

Testing of the full pipeline revealed that bullets and players would go through obstacles. It turned out that the game would sometimes drop bounding boxes - providing an unreliable illusion of mixed reality. Because consistent detection was the highest priority, we implemented a buffer that would keep the bounding boxes for 30 frames before releasing them. This provided a much more persistent playing experience while still providing fast response if an object was moved. While this solved obstacle consistency, it did not address object tracking - meaning that objects could not have specialized abilities because there was not a good system to tell which was the same from frame to frame. A future revision might try to implement an object as a group of bounding boxes whereby a new bounding box is added to a group if it overlaps with the first in the group.

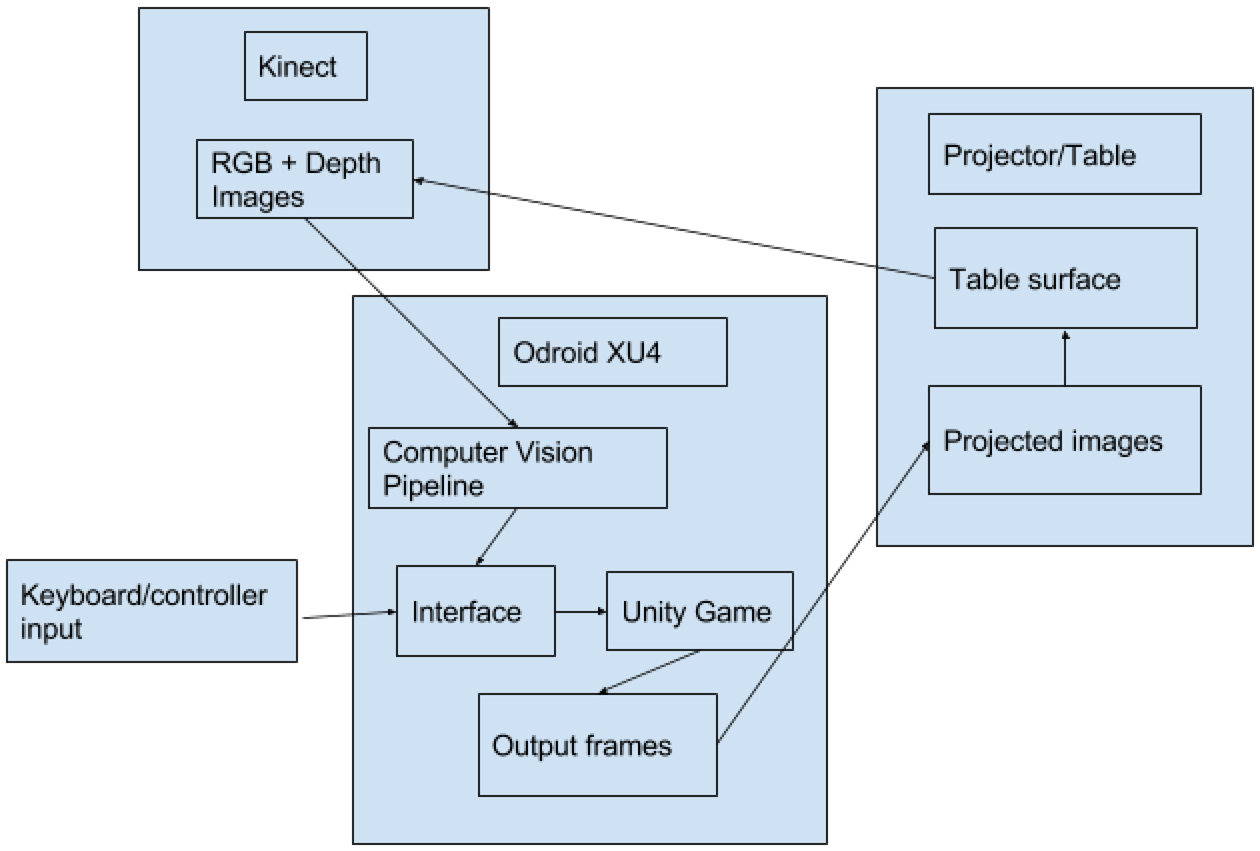
Further player feedback showed the need to reduce speed of the ships because players were mostly dying by crashing into obstacles whereas shooting another ship down took 10 hits. We slowed the ships down and made it take 5 hits to kill, but still liked the idea that the objects were deadly because it made the players respect the object detection system and shifted gameplay from shooting the enemy to positioning your ship around the real world objects. This took the game from a button mashing shooter to more of a game of cat and mouse.Below we can see two screenshots of the in game demo. The first is of a player maneuvering for cover behind a real world object. The second shows a player that has crashed into an object, exploded, and respawned.





A future revision of the demo might explore more experience options, since the system itself is not limited to gaming. Now that the object detection and hardware is in place, a whole range of applications can be implemented - from teaching a kid shapes, to presenting an interactive map, to enhancing board games with animations.

System Diagram



Contributions and responsibilities

Steven primarily worked on the hardware components of the system, assisted Evan with the computer vision algorithm setup and tuning, and the integration of the algorithm and demo applications. For the Odroid, he set up software packages and dependencies, and set up a basic framework for Evan to integrate his computer vision algorithm with. He worked with Evan and Steffen on creating the wooden mount for the projector, Kinect, and Odroid.

Evan worked on implementing the Computer Vision algorithm that would detect the objects on the table. He also worked on a ease-of-use program that would calibrate the Projection image with the Kinect stream to align. He helped in designing and building the wooden mount that held all the components in the right position and angle.

Steffen primarily worked on the design and implementation of the final demo that would showcase the ability of the CV system. He built the demo and iterated over several designs as feedback was given. He also worked on methods to reduce errors introduced by lighting artifacts and the Kinect IPC. He also worked with Evan and Steven on building the wooden mount and integrating the parts of the final system.

Works Cited

"libfreenect2." OpenKinect/libfreenect2, 0.2.0, github.com/OpenKinect/libfreenect2.
"MW820ST." BenQ USA, www.benq.us/product/projector/MW820ST/specifications/.
"ODROID • Index Page." ODROID • Index Page, forum.odroid.com/.
"ODROID-XU4." ODROID WIKI, 27 Nov. 2017, wiki.odroid.com/odroid-xu4/odroid-xu4.
"Open Source Computer Vision Library." Opencv/Opencv, 3.3.1, github.com/opencv/opencv.
Roy, Rob, and Venkat Bommakanti.
magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf.

Project Repository
https://github.com/honeyimholm/Mixed-Reality-Table