Spring 2022

An Accessible Approach to Exploring Space through Augmented Reality

Eden Rorabaugh

An Accessible Approach to Exploring Space through Augmented Reality

Senior Project Submitted to
The Division of Science, Math, and Computing
of Bard College

by
Eden Rorabaugh

Annandale-on-Hudson, New York
May 2022

Dedication

I dedicate this project to my parents, Karen and Pete. Every opportunity I have had, I owe to your sacrifices and hard work. I love you.

Acknowledgements

Keith O’Hara, my SPROJ and academic advisor - Thank you for your help on this project, it would not have been possible without your guidance and support.

Tsitsi Mambo, Samuel Rallis, and Mason Porter-Brown, “The Best Pod” - Thanks for the late night Discord calls and last minute bug fixes. Any team will be lucky to have you all on it.

My Posse - You all have taken care of me for the past four years. You’ll always be my family, and I’m honored to have shared my college experience with you.

My friends - You all saw my project from its earliest stages, and I really appreciate your constant support and encouragement. Thanks for everything.

Abstract

Physically engaging with space is often difficult for people who struggle with mobility. Elderly people and people with disabilities in particular may find it challenging to walk for long periods of time on various terrain in order to explore their environment. This project is designed to provide an alternative way to physically engage with spaces without requiring the user to walk, and I am focusing on the accessibility of Bard's campus specifically. My project involves a map of the college that users can tour in an augmented reality environment. Through the use of a projector-camera system, this program projects a map and tracks objects placed on that map. It tells the user information about the space based on the object's location. Users are meant to collaboratively trace the map and label buildings as they explore them. Finally, users highlight their favorite locations with colored markers, and take a screenshot of the completed map. The colors used are associated with different subjective experiences of the campus and are projected back on the table in the final step of this project. This experience is meant to operate as an alternative to traditional physical tours while also maintaining the communal experience that Bard tours provide.

Table of Contents

Introduction

Motivation

Computer Vision is a way for computers to derive meaning from images and react based on the information they view. It combines artistic and creative elements with standard programming functionalities. The interdisciplinary nature of the field is what originally drew me in. My project specifically utilizes computer vision to construct an augmented reality environment, which, by definition, merges the digital and physical worlds right before a user's eyes. Using a camera projector system, I project an interactive map of Bard onto a large piece of paper on a flat tabletop. The map can be explored using a bright colored post-it note and a separate paper window. Users are meant to trace the projected map, label the buildings, and color in portions of the map they find to be most important. Finally, users are able to take a screenshot of the map and the colors they used are then analyzed and saved in the system.

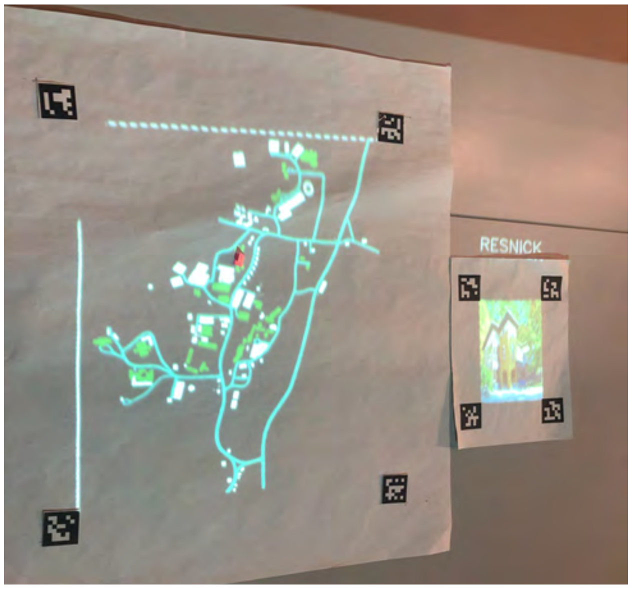


Figure 1. Projected interactive map with multiple paper screens (Author's Image, 2022)

The combination of artistic and technical elements is the perfect application for my Senior Project, especially as a student of the Experimental Humanities. My project acts as a synthesis of several different academic fields. As a computer science major, I rely on programming to make the project possible; however, there are also elements of sociology, studio art, and architecture that I considered in the creation and implementation of my project. For these reasons, it makes users consider the question that many Experimental Humanities projects address: How does technology mediate what it means to be human? In this particular case, the technology I developed connects users with one another in a communal exploration of space. Community is essential in human existence, and I wanted my project to create an experience that brought people together.

I also wanted to pay special consideration to my specific audience since the technology that I am using (i.e. projectors, cameras, Python, etc.) is often daunting and overwhelming for people unfamiliar with its application. I want to combat negative reactions my peers have to technology by providing the college with a computer science project that non-technical people can interact with and contribute to. Making technology accessible is important to me, and creating a project that is exciting and easy to use is my attempt at combating technological inaccessibility.

Finally, I wanted to challenge the inaccessibility of space itself. The parents and grandparents of current and prospective students may find it difficult to engage with the space due to the inaccessibility of our spanning campus. The elderly are not the only group of people who potentially struggle with traversing Bard's 1,000 acres. People facing difficulty with mobility will unfortunately find Bard to be an inaccessible place. The campus is old and outdated with many stairs and hills that make it difficult even for able-bodied people to move about. To combat this, I wanted to create an experience for students, staff, faculty, and visitors to physically explore the campus without over-exerting themselves in the effort. By creating a subjective, augmented reality tour of Bard, users have the ability to see what makes the campus special without walking the vast land that Bard spans across.

My motivation, therefore, is drawn from three different perspectives. First, I am motivated by my concentration since it prompts me to consider how technology affects humanity and how the two interact. Second, I am motivated by the desire to make technology less daunting and more accessible to the Bard community. Finally, I am motivated by the impulse to provide a physical experience of Bard to those that would otherwise be unable to tour the campus in its entirety. Accessibility, both emotional and physical, is important in cultivating the inclusive space that Bard strives to be. My project is meant to continue pushing the college towards widespread accessibility, and that is why it is important.

Related Work

DynamicLand

Founded by Bret Victor and Alan Kay, DynamicLand has been a space for creativity and collaboration since 2018. It is marketed as a “new computational medium where people work together with real objects in the real world.” This description reveals the non-profit’s emphasis on human interaction through the shared use of their “communal computer.” The warehouse DynamicLand resides in is filled with cameras and projectors that read colored dots on pieces of paper and project programs associated with the dots on those papers. Users are not confined to screens and cubicles to engage with this form of computational media. They are encouraged, and essentially required, to verbally and physically engage with others in the process of interacting with DynamicLand. This project is an excellent example of technology encouraging users' humanity. It prompts people to be physical in their thoughts through actions like moving papers and touching cars. They can experiment and improvise which is difficult to fully accomplish behind a screen programmed for millions of users. In DynamicLand, not only do users have endless creative agency, but they also have the ability to interact with this technology without sacrificing their humanity.

DynamicLand is a clear example of what I consider to be a healthy interaction between humans and technology. For that reason, it intersects well with my Experimental Humanities concentration. As an EH student, I have to constantly ask myself how technology mediates what it means to be human and how the technology I develop affects people. This specific technology, DynamicLand, creates an environment where humanity is encouraged above all else. Technology is therefore used as a medium through which humanity can be presented. I want my Senior Project to emulate this idea of technology uniting the people that use it, and I want to incorporate the physical ability that DynamicLand presents to its users.

Figure 2. Users interacting with DynamicLand [9]

SyMAPse

In her SyMAPse project, Julia Chatain and her team constructed a digital mapping tool using a projector and two cameras. The cameras were used to detect paper movement, hand gestures, and object placement while the projector displayed the map based on these variables. The team used integrated and non-integrated 3D and 2D objects for their augmented reality interactions. Integrated objects represent augmented aspects of the map such as buildings and landmarks. Non-integrated objects present users with data about the location, such as population density. In their paper, Chatain and her team discuss their desire to incorporate both physical and virtual elements into their project since they wanted to keep “most of the features of both the mediums” [12]. Users are expected to express themselves using pens, removing the technological constraint.

Like DynamicLand, SyMAPse provides users with an environment that supports the physicality of humanity without applying limitations. This is the kind of freedom I want my own project to emulate. Users of my Senior Project are able to add their own personal touches to the Bard maps through drawing, and they can keep that map with them after they finish the AR tour. SyMAPse also presents users with the ability to explore space specifically through movement, object placement, and art. This representation of location is present in my own project, since space is the main topic that my augmented reality environment explores.

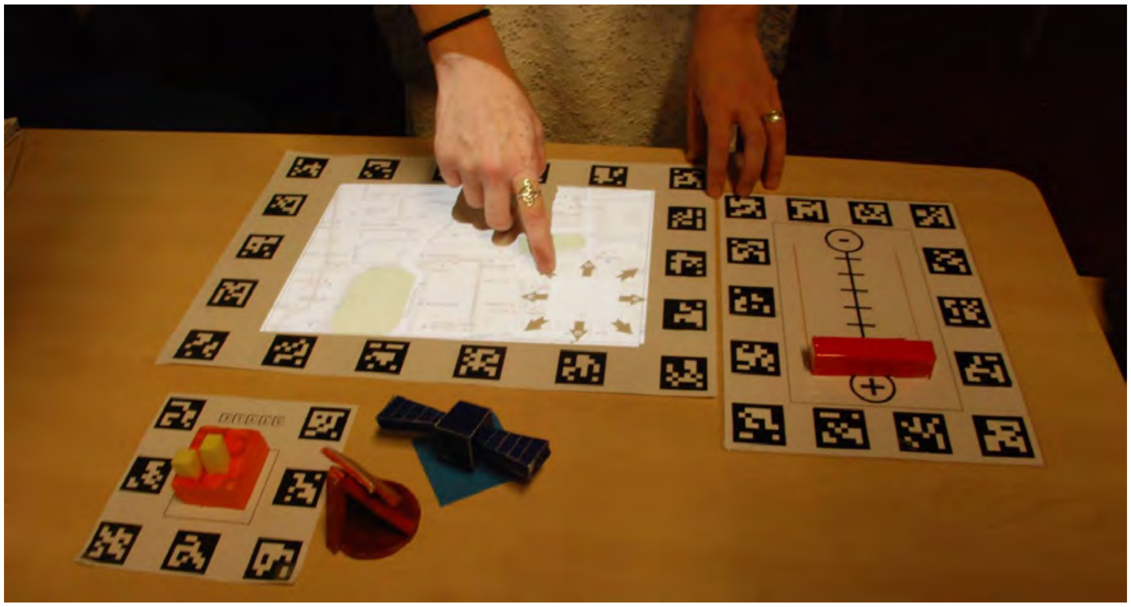


Figure 3. Hand gestures and objects are used to interact with SyMAPse, and multiple sheets of paper can be used in these interactions [12]

Subjective Maps

Typically, when one looks at a map, they are utilizing it to determine direction and location; however, maps can also serve as a vehicle for storytelling. Subjective maps are used to present a personal experience the creator has with the space depicted. For example, the map presented below is a subjective map of Rennes, France, constructed by children living there. This kind of map is not ideal for objectively providing information to readers, but is great at revealing relationships between people and the spaces they inhabit. For example, the children marked places on the map like “Traffic jams street” and “The frogs’ choir area” [12]. By providing a subjective representation of their home, the children are able to take space to a more personal level. It is no longer viewed just as a set of roads to travel from one location to the next, but is rather depicted as a dynamic environment where memories reside and experiences take place. It presents space as something that people can have a relationship with, rather than simply use.

In my Senior Project, I want the experience of Bard students to be represented. Since my project acts as an augmented reality tour for prospective students and their families, I want the individual experience of students to be present in the tour, just as the subjective experience is provided by tour guides on physical tours. In order to incorporate this element into my project, I include the ability for users to add their own personal contributions via colored sharpies. When users trace the map projection, colored markers correspond to different building preferences that are then archived to understand the relationship users have with buildings across campus. This gives the student body the ability to present their experiences of the campus and show that it is more than just a space to inhabit. Bard’s campus is unique in many ways that have emotional impacts on students. From the natural beauty of Blithewood, to the academic environment that the RKC provides, the setting affects students, and I want that effect to be represented in my AR environment.

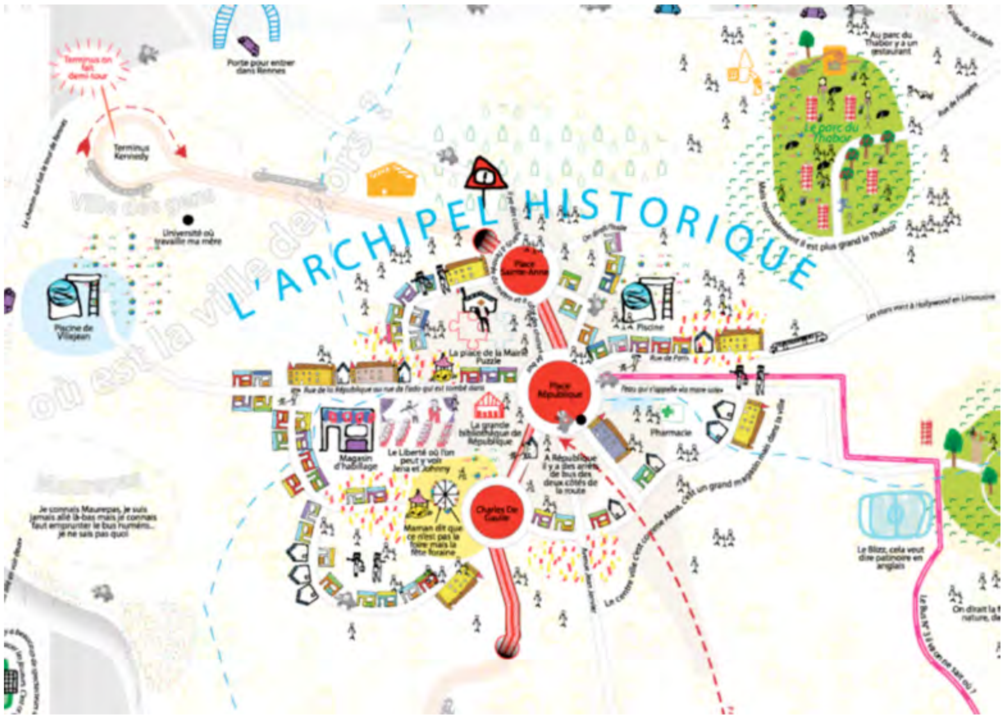


Figure 4. Subjective map constructed by the children of Rennes [12]

A Mixed Reality System for Learning Data Structures

In his 2020 Senior Project, Cullen Drissel constructed an augmented reality environment, similar to the one that I am working with, using a projector-camera system. In his project, he designed a program that would project a spanning tree based on the distances between wooden tokens taken in by the camera. He also incorporated the ability to detect and analyze text into his project, which is a unique and useful addition. My project is a partner project to Drissel's, as his was the original inspiration for my project. I am following a similar route to Drissel, but adding a more visual element to the final result by basing mine around a map. Nevertheless, his project revealed the novelty of projector-camera systems and inspired me to utilize such a system in my project. I am following in his footsteps by relying heavily on the OpenCV library, and I have found it useful to have an example Senior Project to base mine around.

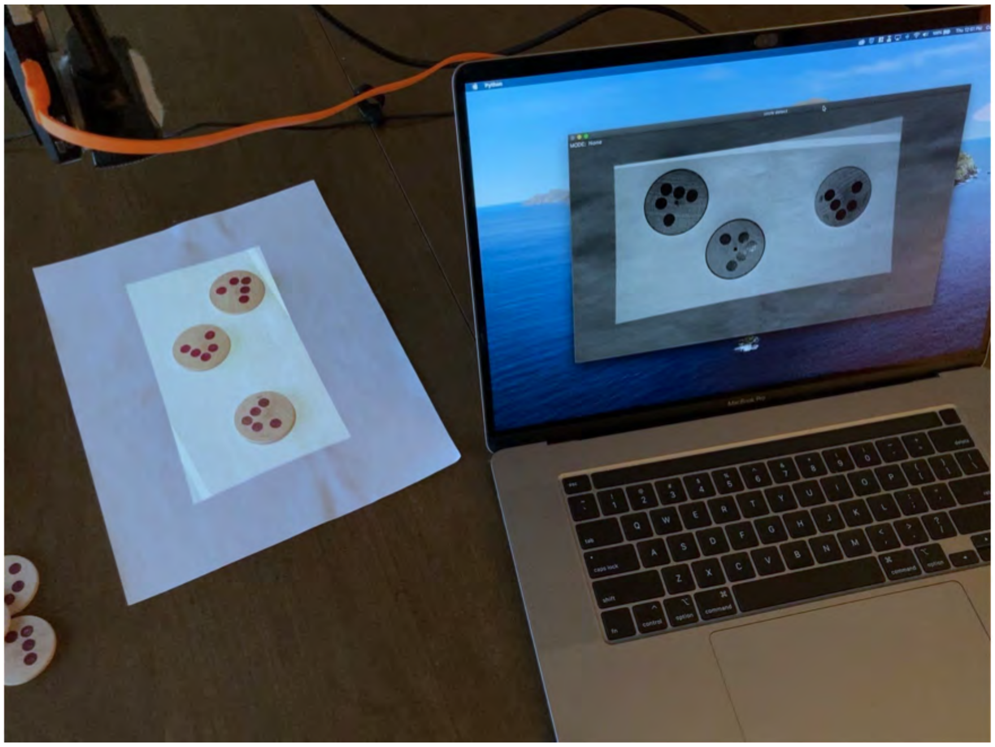


Figure 5. Snapshot of the early steps in Drissel's project [6]

An Augmented Workplace for Enabling User-Defined Tangibles

In their paper "An Augmented Workplace for Enabling User-Defined Tangibles," Markus Funk, Oliver Korn, and Albrecht Schmidt create an augmented reality system where users can determine what objects are included in the system and how those objects will operate. According to the authors, they "introduce an interaction concept for creating bindings using physical objects as a tangible control and subsequently unlinking objects from the digital function again" [13]. This gives users maximum control over how the system works since they are able to determine what is included in the system and how it is used. The paper discusses different ways that user-defined tangibles can be tracked. For example, marker identification is ideal for camera projector systems that are built into tables. The camera is able to recognize ArUco or fiducial markers that are placed on the bottom of the object for tracking purposes, though this does present limitations. The use of “bottom mounted markers requires the surface to be sensitive” so that when an object is placed on the surface, the marker can be distinguished [13]. Their system uses feature-based identification to determine what objects are associated with function bindings. Users are therefore able to define tangibles by placing them in a “pre-defined area on the workspace” after which the “top-mounted camera takes a picture of the object,” and eventually, the “depth data of the object is captured to learn the object’s shape” [13]. After defining the tangible, users are able to choose functions to determine how the tangible will be controlled, and can then use the object as they defined it. Unlinking is the final step, which is only done to “turn a user-defined tangible into a normal object again” [13].

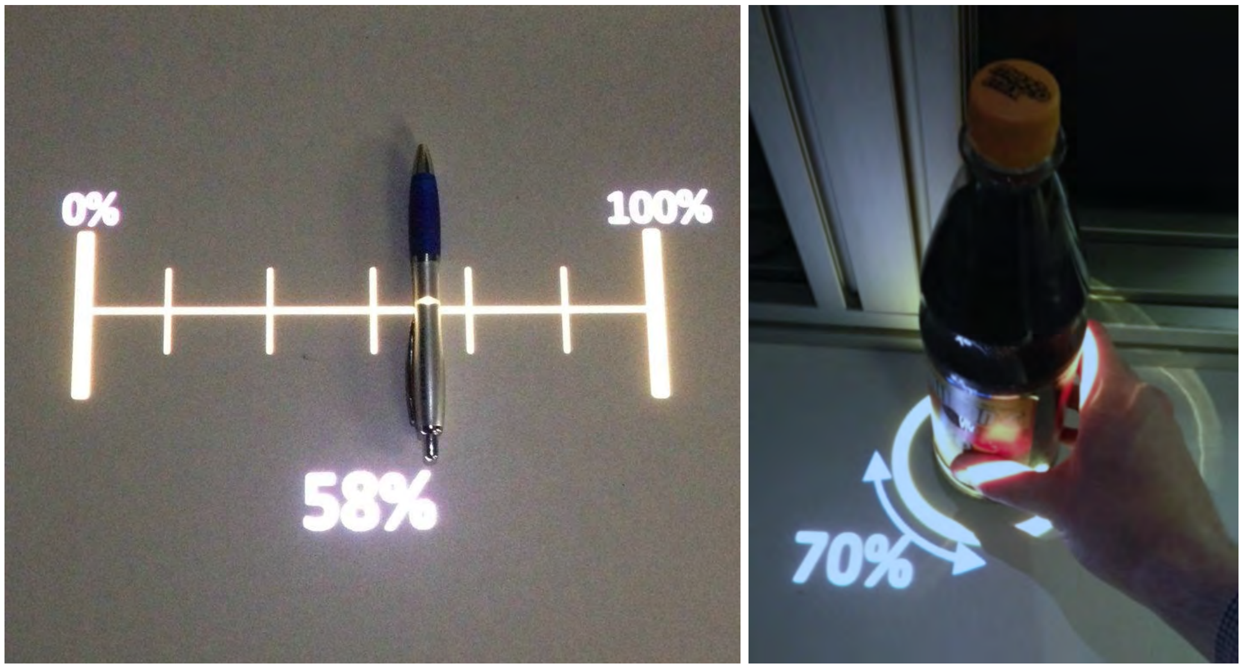


Figure 6. Examples of user defined tangibles [13]

The concept of user defined tangibles is particularly applicable to my project because it provides the user with control over the environment they are exploring. When users provide their own tangibles, they are able to physically connect with the system in a personalized way, making their experience all the more impactful. Though I do not include a user defined tangible component in my project, I do focus on the subjectivity and control of the user throughout their experience. Users are able to reposition and alter the ArUco paper screens and are able to use detection objects to explore space. By making users feel in control of their environment, I hope to make the experience less technologically overwhelming and more personalized.

Interaction in Augmented Reality: Challenges to Enhance User Experience

Yahya Ghazwani and Shamus Smith explore how the user, the user interface, and the virtual content interact with one another in AR systems. Augmented reality, defined as combining virtual and real content, registering in 3D, and interacting in real time, “sits within the mixed reality space that connects the real world and the virtual world” [16]. They discuss the variations of user, interface, and content combinations that can be present in an AR environment ranging from passive users in solo, offline environments to active users in collaborative TUI based environments.

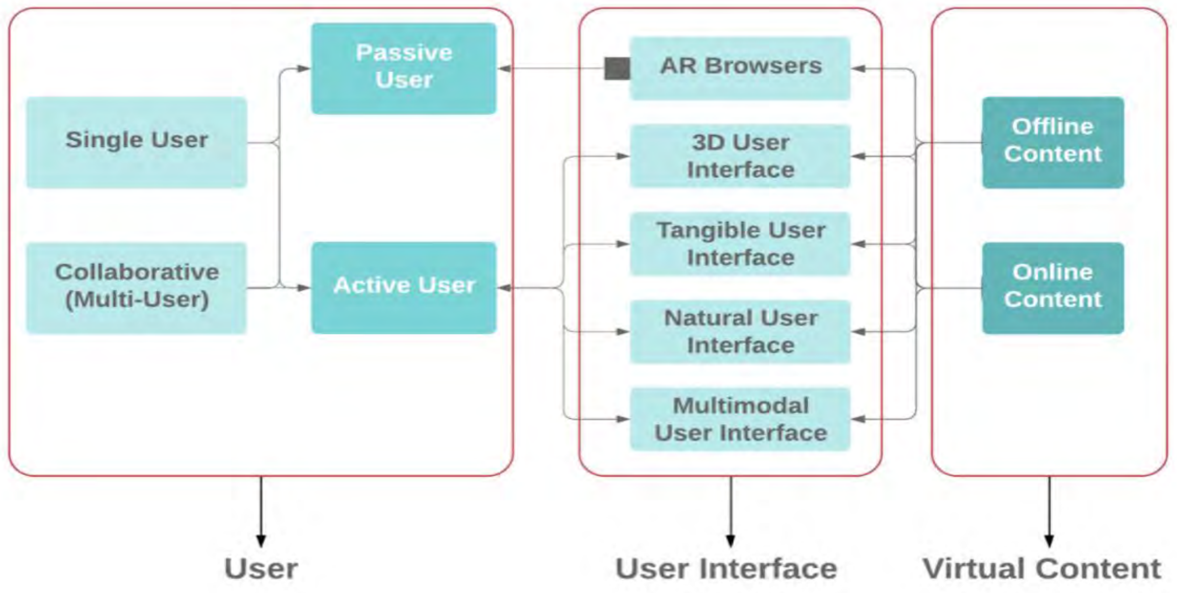


Figure 7. Diagram exploring the options AR systems creators have when defining the environment [16]

By exploring the challenges and benefits to each kind of environment, this paper helped me determine exactly what kind of system I wanted to create. I wanted my environment to allow *active* users, meaning users can "select, manipulate, edit, add or delete virtual content" [16]. I assumed that users who play an active role in the system have a better sense of content engagement, and this paper confirms that understanding. I also wanted my environment to be collaborative. According to Ghazwani and Smith, "collaborative environments are when virtual information is shared amongst different augmented reality system users" [16]. I want my project to allow for collaboration between different users within the system, but also want that interaction to be done offline. No network should be required and no user should be able to interact with my system without being physically present in the space. Since my project focuses on tangible interaction, it is imperative for users to be physically engaging with the environment. Since I will also be utilizing Tangible User Interfaces (TUIs) in my project, this paper was useful in considering the limitations that TUIs present. They restrict a user's ability to multitask and restrict the interaction area. I find that these limitations exist when interacting with any tangible object outside of an AR environment, and therefore are able to mimic reality in a way that does not impede the success of my project.

Background

OpenCV in Python

This project relies on a Python based interface to OpenCV [14]. OpenCV combines computer vision techniques such as object detection, color tracking, contour determination, and more, into a single, easily accessible library. It can be used in multiple programming languages, including Processing, but I decided to use a Python based implementation. The OpenCV documentation for Python is robust, making it easy to apply. The utilization of Python itself also stems from a desire to provide readable code for future progression of this project. Ideally, this is a project that can remain useful to Bard long after I graduate, and using a language that is easy to understand makes the transfer of project management much faster. For this reason, I have also made a point of heavily documenting my project progress.

Homography Matrices

Computer vision and augmented reality often depend on homography matrices. Homography matrices provide maps between two planes, or more specifically in this project, between two images. When two different points are selected to determine a homography, one point must undergo a series of transformations to mirror the other and vice versa. The homography matrix is the matrix containing the transformation instructions from one plane to the other and is essential when determining how to properly project on a flat surface based on camera inputs.

$$s\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 8. The above equation relates the homography between two planes [14]

If two images represent the same plane from different angles, then they are related by a homography matrix [17]. For example, in my projector-camera system, both my projector and camera are depicting the real world environment on the same flat plane from slightly different angles. Though they can be set at different angles, their fields of view must overlap so that they are representing the same view; otherwise, no homography can be estimated between the two images. For example, if my projection image is a map of Bard, but my camera input is a blank wall, there are no points to determine similarities between the two images. If, however, my projection image is a map of Bard and my camera input is that projected map from an extreme angle, then a homography can be determined.

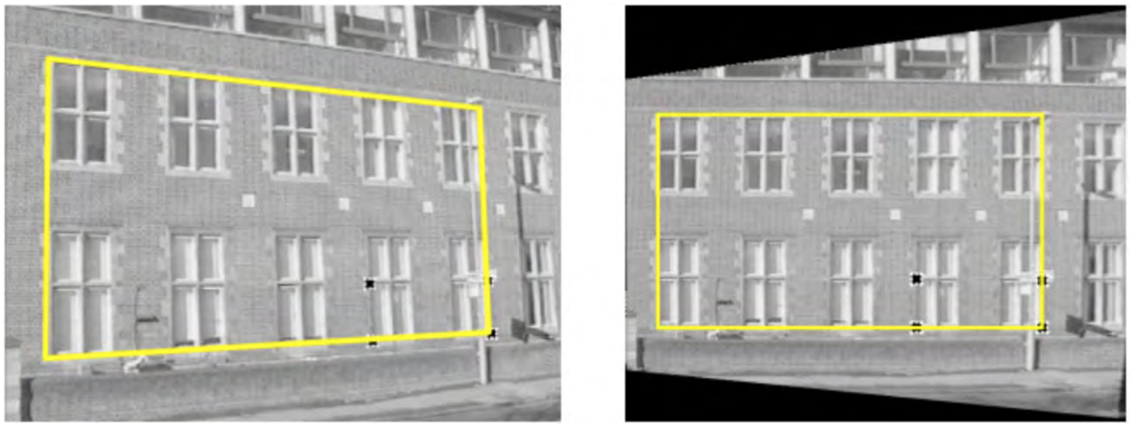


Figure 9. Two images taken from different angles where a homography matrix is determined [14]

Determining a homography matrix is made simple with OpenCV’s *findHomography* method. This method requires two sets of points with at least four correspondents that the matrix should be drawn between, and it outputs the matrix itself. Without a proper homography, my projection would be warped, making interaction impossible. Calibration is the first necessary step in my project so the original grid of the projection and camera scope can be determined and mapped to one another. Separate homographies are necessary when determining other objects within the overall projection screen. For example, the ArUco paper screen requires a homography outside of the calibration matrix so that a projection can occur specifically within the bounds of that particular paper.

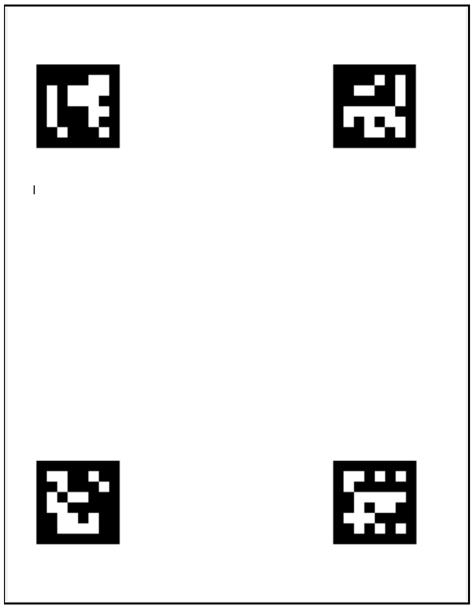


Figure 10. An example of an ArUco paper that my campus map could be projected on (Author's Image, 2022)

While OpenCV's *findHomography* determines the homography matrix between two sets of points, the *warpPerspective* function alters the image to apply the transformations of the homography matrix. It takes in the image undergoing the transformation, the homography matrix, and the size of the desired output image. This function is required in projections so that the projected image is not applied to the camera coordinate system, but to the coordinate system of the projected image on a flat surface.

NumPy Arrays

NumPy is a Python Library that provides array objects that are up to 50 times faster to process than standard Python lists [18]. NumPy arrays have *locality of reference*, meaning that while Python Lists are arrays that reference pointers to objects, NumPy array objects are stored in a contiguous memory block with no pointers. Since the array is stored in a "data buffer" or a homogenous, constant memory block, rather than being scattered across the system's memory, programs are able to access NumPy array objects at considerably faster rates [5]. This speed is necessary in my project since arrays are used to determine points in an image plane. I use NumPy arrays to construct blank windows for projection, to create homography matrices based on image points, and to store my ArUco marker objects when detected. Since computer vision relies so heavily on matrices, and since my project needs to be as fast as possible for accurate projection without delay, NumPy arrays are essential.

Though NumPy arrays are necessary for my particular project needs, this does not mean that they can simply replace Python lists as a whole. While NumPy arrays are faster in accessing points in the given array, they are slow in appending or deleting objects from these arrays. This is because additions and deletions require a change in the data buffer size and result in a changed memory allocation for the array itself. Python lists, though spread across a system's memory, can add or delete new items much faster since they do not rely on an allocated data buffer.

ArUco Markers

ArUco markers are "synthetic square markers composed by a wide black border and an inner binary matrix which determines its identifier (ID). The black border facilitates its fast detection in the image and the binary codification allows its identification and the application of "error detection and correction techniques" [14]. The markers can be processed by OpenCV algorithms regardless of positioning based on the dictionaries of markers usable in OpenCV. These markers are commonly used in augmented reality environments since they are fast and easy to detect. The marker's ID and corners can be determined by the *aruco.detectMarkers* function. With distinct outputted corners and IDs, projection within the polygon made of ArUco markers is simple and exactly what I want for my project. The markers can be made very small and can be taped to any flat surface which makes it easy to resize my desired projection window. Their efficiency, flexibility and speed are why I settled on ArUco markers for detecting paper screens in my final project.

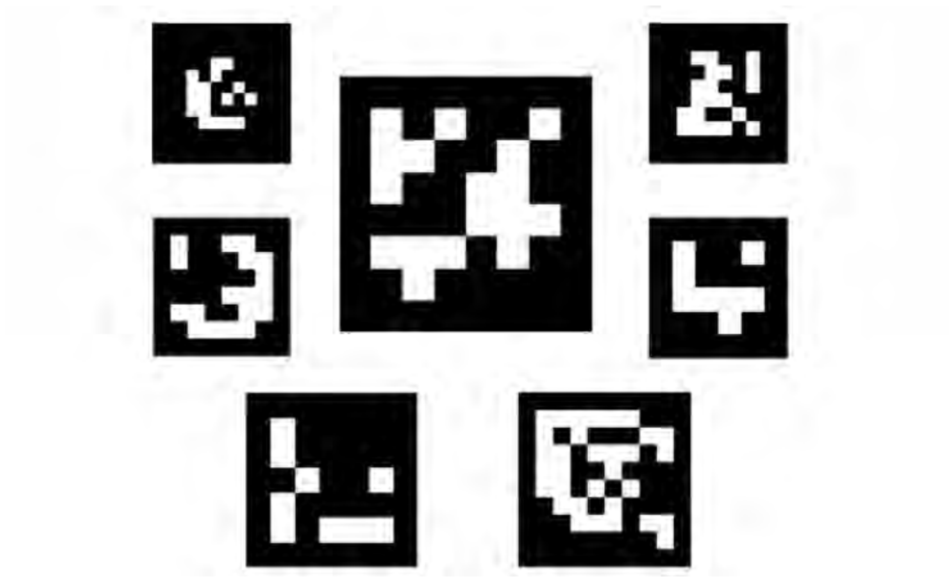


Figure 11. Examples of ArUco Markers [14]

Projector-Camera Environment

I experimented with several different cameras, projectors, and locations before deciding on the final set up using a Logitech webcam and an Optoma short throw projector in the Experimental Humanities shipping container. The Logitech camera was easy to move, and its feed could be manipulated using a Mac application called Webcam Settings. This application allowed me to specify the focus of the camera so that moving objects in the frame did not disrupt the plane it was specifically meant to detect. I could also change aspects of the image input such as brightness and contrast to provide the clearest image intake despite dynamic lighting conditions. The Optoma short throw projector was my final demonstration projector because it can be mounted incredibly close to the surface it is projected on yet still illuminate a large portion of that surface. This makes mounting more portable and simple since the distance from the projector to the projection surface can be much shorter than the distance of a normal projector.

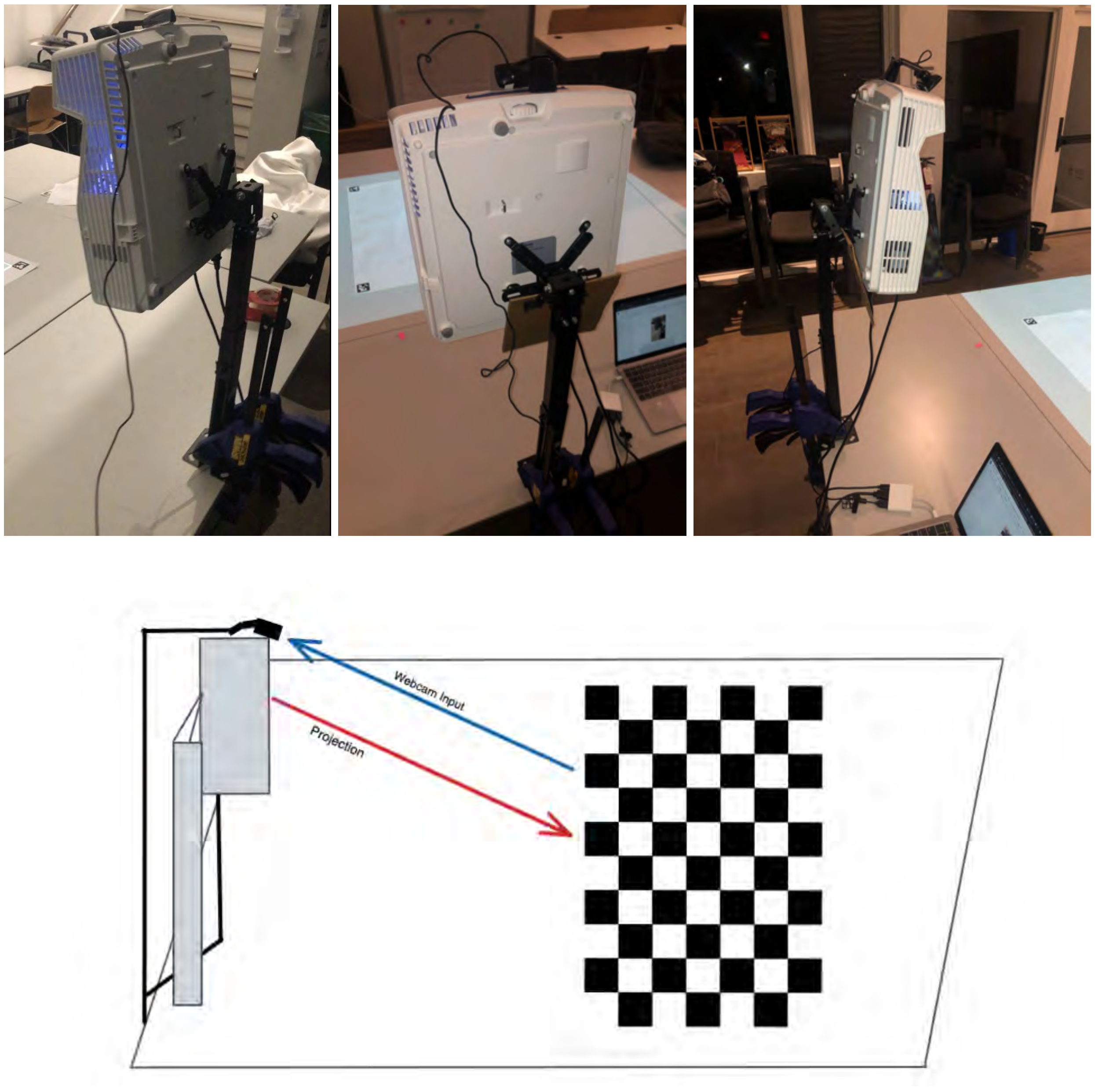


Figure 12. Mounted environment setup with Logitech camera taped to the top of the Optoma projector held to the tabletop with hand clamps (Author's Photo, 2022)

Since I wanted my project to be mobile, the actual mounting of the system was important for maximizing mobility of the system. The Optoma projector is meant to throw a projection against a vertical surface rather than a horizontal one, so I had to construct a tabletop mount that allowed it to face downwards. The mount's stabilization required the use of hand clamps that could secure its base firmly to any table with a strong enough edge. The tables in the Experimental Humanities building proved to be perfect for this task. They are easily moved and reorganized to quickly assemble this project environment without any permanent installation. Since my project is categorized as an Experimental Humanities project, it seems fitting that it be demonstrated and constructed in the shipping container itself. Though light conditions proved to be inconsistent and, at times, disruptive to the success of this project, the ease and mobility of my project environment made this location ideal.

Methods

Projector-Camera Calibration

The first step to creating an effective projector-camera environment is calibrating the camera and projector. While the camera view and projector window overlap, there is often a certain level of distortion between what the camera sees and what the projector shows. Essentially, camera calibration recovers the mapping between a point in the projection and a point viewed by the camera [15]. It determines the homography matrix between the point in the 3D projected world and the point in the 2D plane seen by the camera. Since all the observed points are on a flat surface, every point can be mapped to another using transformations. As long as neither the camera nor the projector are moved at all after the calibration step, this transformation will remain the same throughout a session using this program.

My calibration method, *calibr.py*, takes in three parameters to determine a homography between the camera and projector. First, it takes in the image from the camera, *vidPic*, which will be used in the OpenCV *findChessboardCorners* function to find the corner intersections in the projected image detected by the camera [14]. It then takes in *patternPoints* which is determined by the intersections detected by *findChessboardCorners* on the input pattern image in our main method. Finally, it takes in a boolean, *calibrated*, to determine if the calibration is successful. Since the *calibr.py* method is called after the calibration pattern is shown by the projector, the *if* statement is accessed. Corner intersections should have been identified in both the camera and the pattern image. Homographies, *homogCamToComp* and *homogCompToCam*, are then determined and returned. Two homographies are computed, one representing the translation needed from the camera to the projected image and the other going from the projected image to the camera. The *calibrated* boolean is also returned to show that calibration was successful. These homographies are used when the projection is updated and they ensure the mapping is successful and accurate on the determined matrix plane.

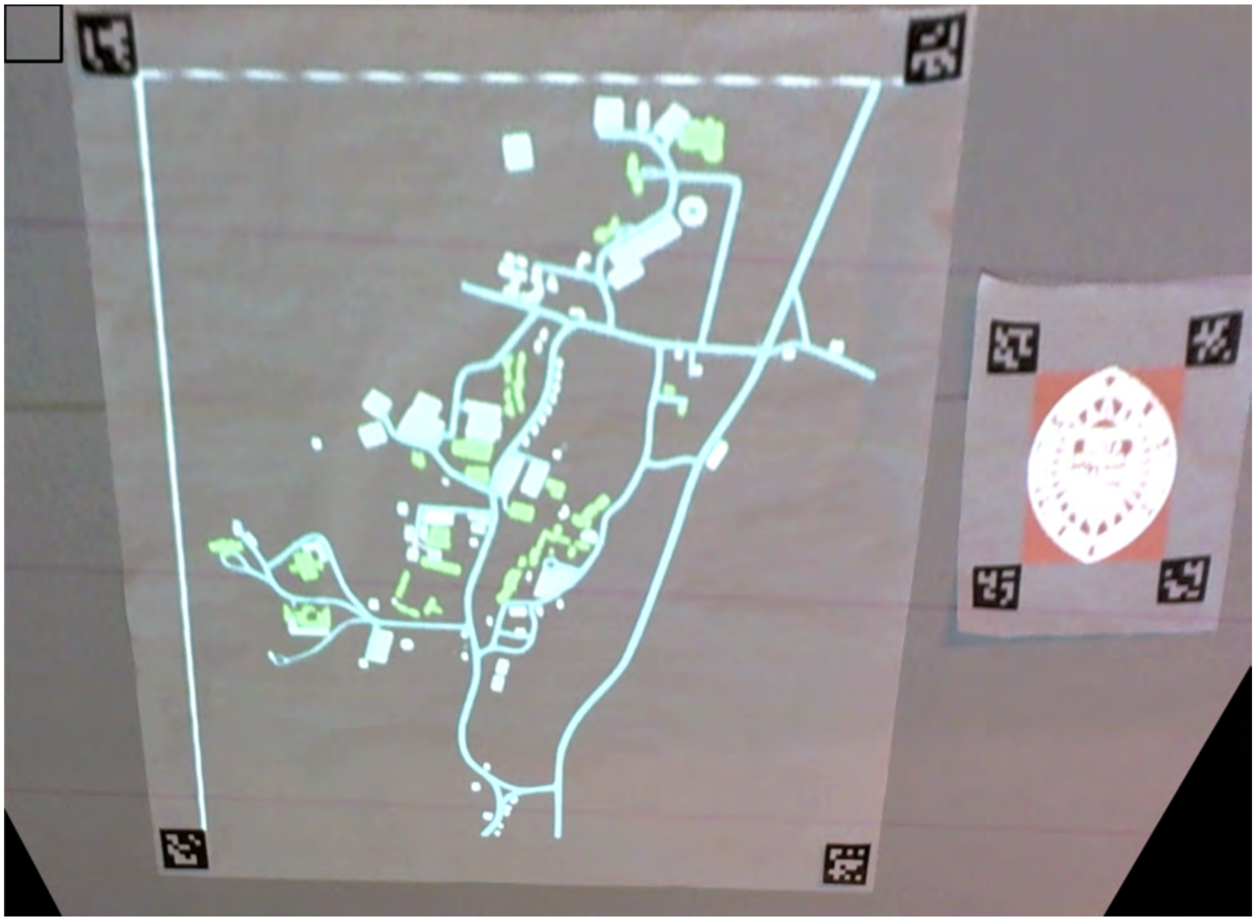
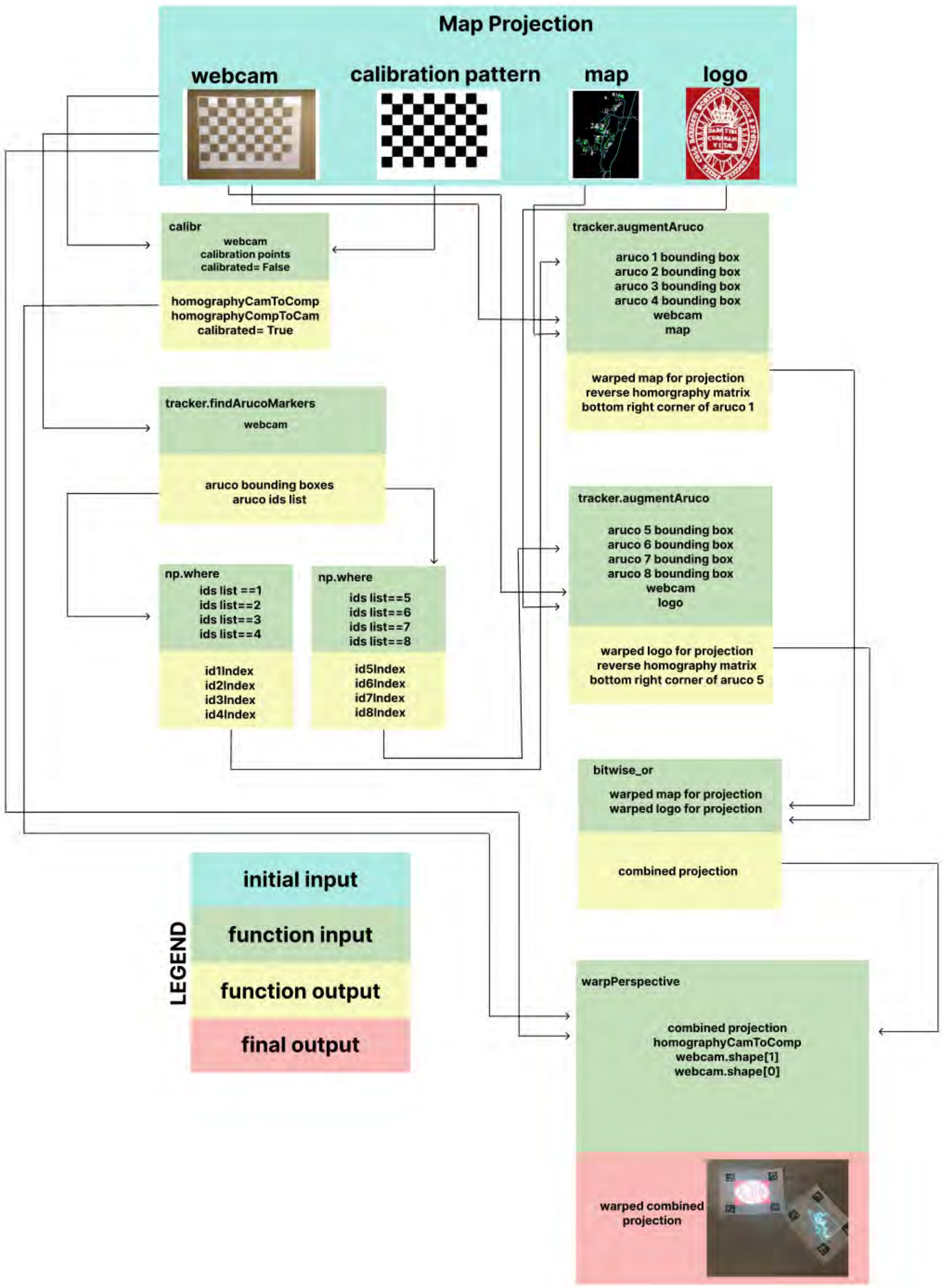


Figure 13. Warped perspective on the *homogCamToComp* homography matrix (Author’s Photo, 2022)

ArUco Projection

After performing the calibration step and retrieving the homography matrices, the image taken in by the webcam must undergo a *warpPerspective* by the *homogCamToComp* matrix so it properly represents how the image is manipulated from the camera to the projection window. This is done to properly exhibit how the camera detects the 3D space from its perspective and applies the transformations so that the unwarped image is passed into the method. After the transformation step is completed, this method uses ArUco markers to determine the space the projected image should be displayed in. The *tracker.py* file is referenced to find the ArUco markers within the webcam's frame. The *tracker.findArucoMarkers* function used in my *tracker.detectMarkers* method takes in the input *img*, which in this case is the webcam input. The other parameters of the method are set since I only use the 6x6_250 ArUco dictionary. The *aruco.detectMarkers* method determines which objects within the frame are actually ArUco markers and returns a list of the bounding box and IDs of these markers. After markers have been found, I use the bounding box arrays that are attached to each individual ArUco ID, and construct a point array of ArUco ID 1's bottom-right corner, ArUco ID 2's bottom-left corner, ArUco ID 3's top-left corner, and ArUco ID 4's top-right corner since this is the space within the paper I want to project on. This point array is only constructed if ArUco markers 1, 2, 3, and 4 – ArUco A– are detected, and I construct another point array if ArUco markers 5, 6, 7, and 8 – ArUco B– are detected where pictures of campus are projected.


Map Projection
webcam
calibration pattern
map
logo
calibr
webcam
calibration points
calibrated= False
homographyCamToComp
homographyCompToCam
calibrated= True
tracker.findArucoMarkers
webcam
aruco bounding boxes
aruco ids list
np.where
ids list ==1
ids list==2
ids list==3
ids list==4
id1Index
id2Index
id3Index
id4Index
np.where
ids list==5
ids list==6
ids list==7
ids list==8
id5Index
id6Index
id7Index
id8Index
tracker.augmentAruco
aruco 1 bounding box
aruco 2 bounding box
aruco 3 bounding box
aruco 4 bounding box
webcam
map
warped map for projection
reverse homorgraphy matrix
bottom right corner of aruco 1
tracker.augmentAruco
aruco 5 bounding box
aruco 6 bounding box
aruco 7 bounding box
aruco 8 bounding box
webcam
logo
warped logo for projection
reverse homography matrix
bottom right corner of aruco 5
bitwise_or
warped map for projection
warped logo for projection
combined projection
warpPerspective
combined projection
homographyCamToComp
webcam.shape[1]
webcam.shape[0]
warped combined projection
LEGEND
initial input
function input
function output
final output

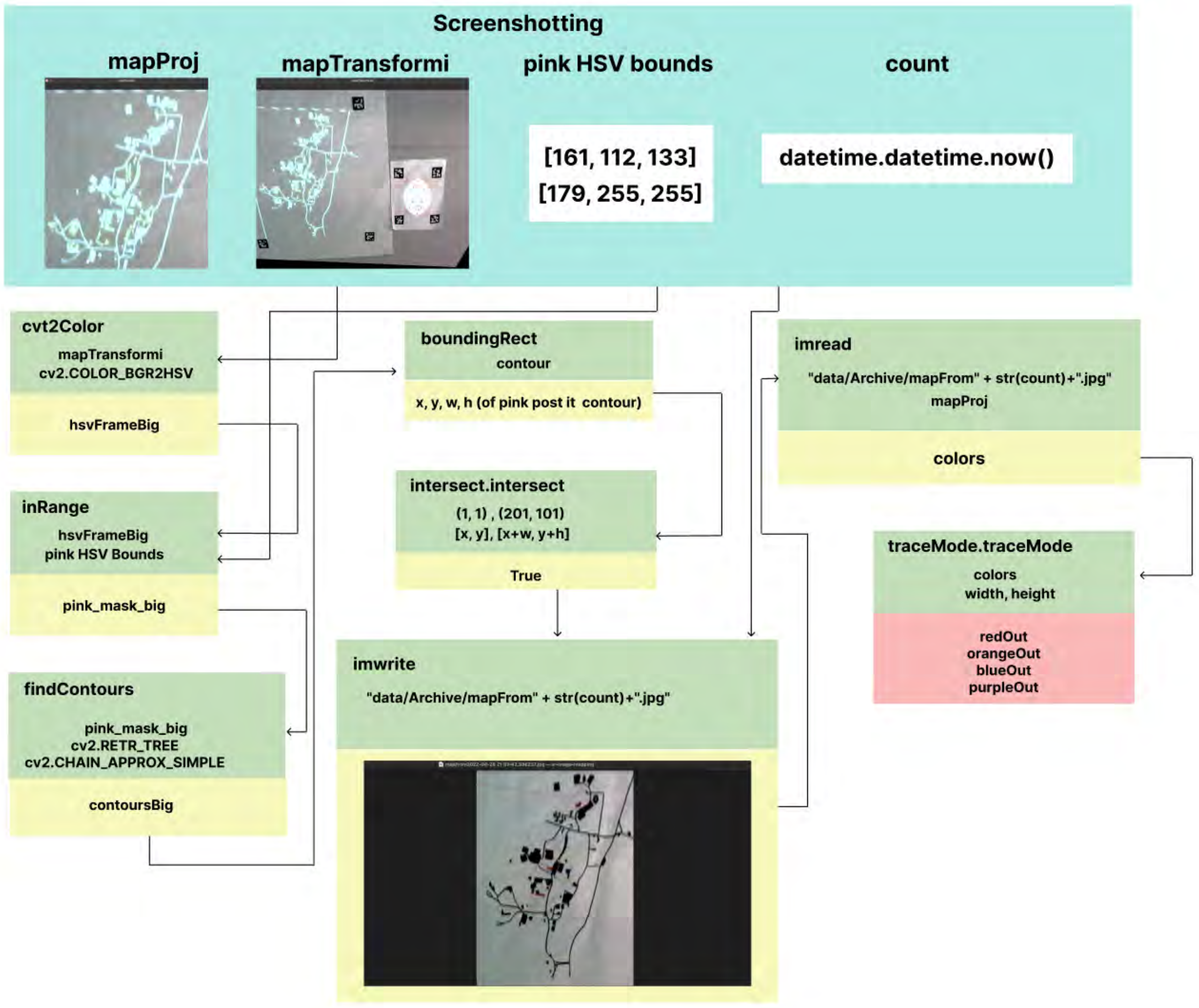


Figure 14. Basic wireframe diagrams of calibration and map projection program and screenshotting (Author's Image, 2022)

After the edges where the image will be projected are determined by the ArUco markers, a homography between the projection image and the physical ArUco corners is determined by the *tracker.augmentAruco* method which takes in the four identified markers, the webcam input, and the image to be projected. That homography matrix is finally used to project the image back onto the physical ArUco paper. I utilize both the *findArucoMarkers* and *augmentAruco* twice since there are two ArUco paper screens that I use in the application of this project. ArUco A is a large blank sheet where the map of campus is projected and ArUco B is a smaller blank sheet that depicts the Bard Logo when no location is selected on the map. If a location is selected, ArUco B shows images of the selected building.

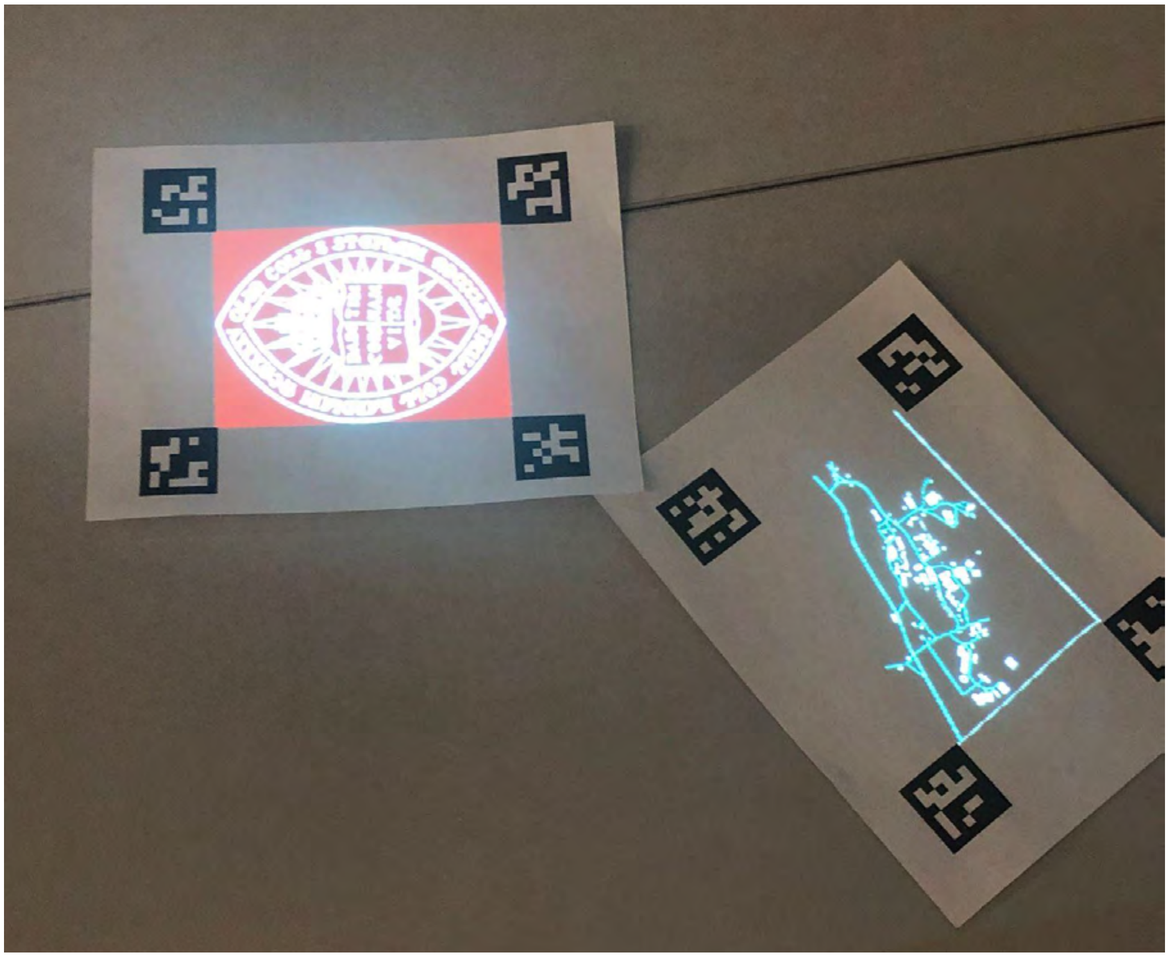


Figure 15. ArUco B projecting the Bard logo (left) and ArUco A projecting the campus map (right) (Author's Photo, 2022)

I use a series of if statements to determine which projection to show. If only ArUco A is detected, the campus map will be projected alone, and if only ArUco B is detected, the Bard logo will be displayed alone. When both sets of ArUcos are detected, I use OpenCV's *bitwise_or* function to combine the two images so both projections can be represented. The *bitwise_or* function essentially takes two matrices, in this case the projection of the map and the projection of the logo, and outputs a colored pixel for each of the corresponding bit positions that have a colored pixel in either or both of the inputs. This only takes place if all eight ArUco markers are detected within the webcam frame, and only when both are present will ArUco B change based on the colors detected on ArUco A.

Map Sectioning

My project relies on the sectioning of a Bard map in order to determine which spaces on campus the user wants to explore. For this reason, it was necessary to separate the map into sections for each of the main locations on campus. By showing the map image in a window, I was able to use my mouse to determine which points make up certain portions of the campus. By clicking on corners of buildings and locations, I was able to determine their bounds. I determined sectioning around the top left and bottom right corners of the locations I wanted to highlight in my project, and I assured the accuracy of these points by displaying them on my *mapProj* image which depicts the webcam footage of the projected image within the ArUco markers detected, rather than simply displaying these rectangles on the image itself.

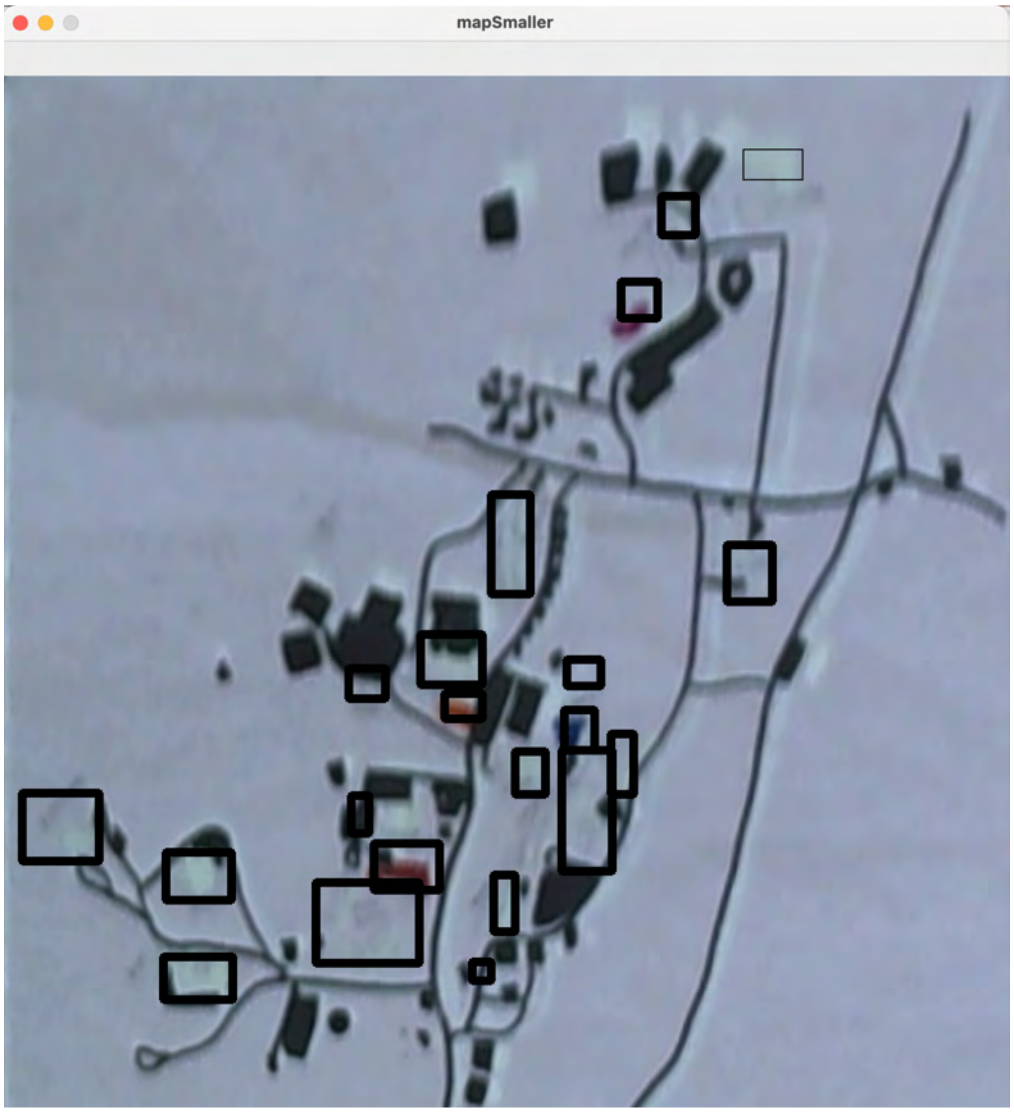


Figure 16. Example of a *mapProj* screenshot with the rectangle outlines depicted (Author's Image 2022)

These arrays are used in my project to determine when colored objects are placed within their bounds. If the location of the detected object overlaps with the array, then that map location will be considered selected and another window will appear displaying an image of the selected location on ArUco B.

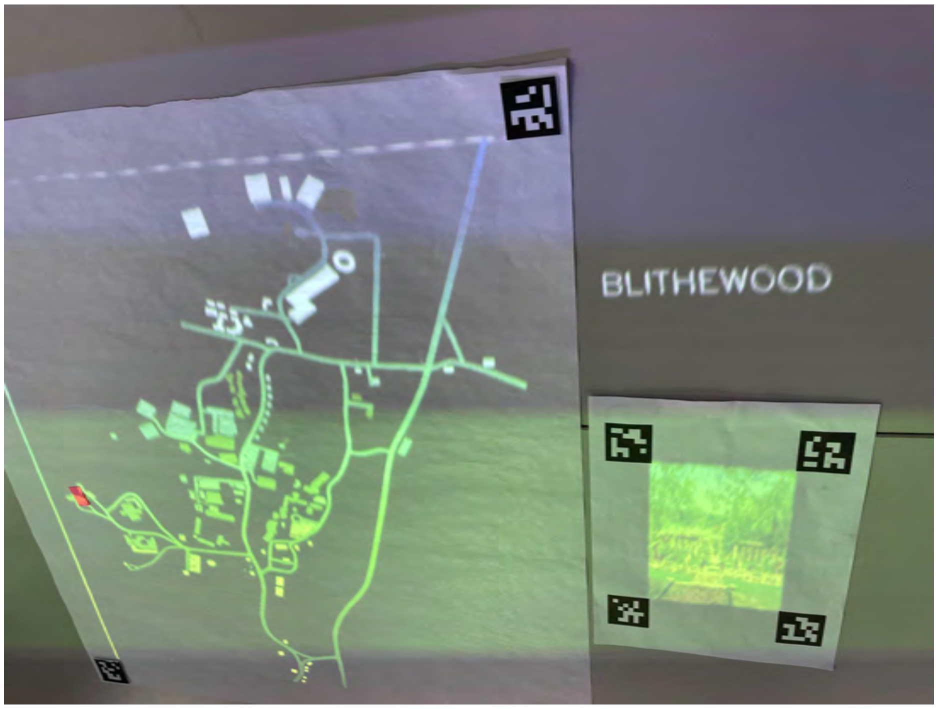


Figure 17. Blithewood has been selected on the large map using the pink post-it and an image and title for Blithewood are projected in the smaller ArUco window. (Author's Photo, 2022)

Color Detection

Selection capability is essential in this project since portions of the map must be selected in order to be explored. In order to incorporate selection in this project, color detection is necessary so that the specific selection tool can be recognized. I originally created a method for tracking based around a laser pointer. Since laser pointers project a bright beam of light at a plane, they are easy to track using OpenCV's *minMaxLoc* function [14]. The function determines the brightest point in an image, which is always the laser pointer. Though this program was successful, I ultimately shifted to color tracking since laser pointers are not as readily available as normal brightly colored objects. Prototyped in my *color-tracking.py* program, I began by determining the upper and lower bounds of the color I want to track. If I am tracking the color

red in a range of hues, the bounds will be vast, but if I am detecting a specific red shade, the bounds will be much slimmer. I construct a mask from the color detected in the image within the bounds provided. If the area of the mask is large enough to overcome the threshold, then it is considered to be present in the image. In my prototype code, a projected box appears around the colored object with words defining the color detected. Implemented into the overall project, color detection works as laser detection does. When a color is detected over a certain area on the map, information about that area will be revealed. The bounds of the map are once again defined by the map sectioning discussed
previously.

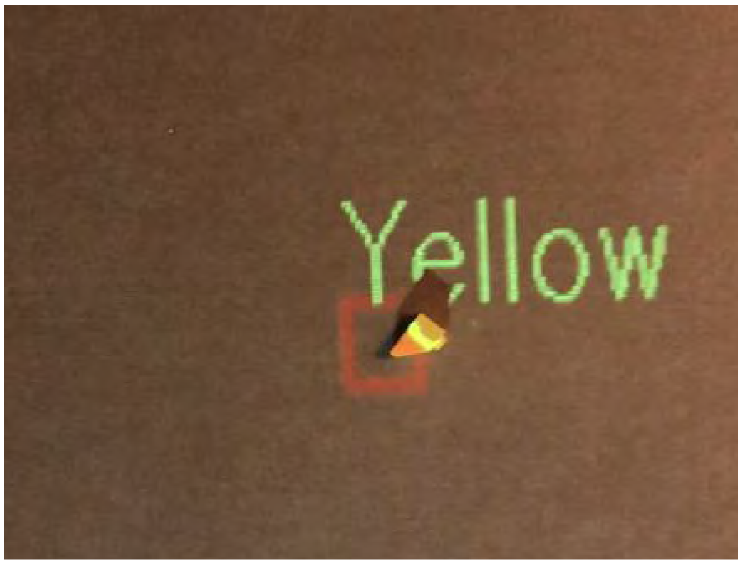


Figure 18. Color tracking prototyping with projection of bounding box around color (Author's Photo, 2022)

In the final implementation of my object color tracking, I utilize a public program on GitHub by Federico Ponzi, a Systems Developer Engineer at CloudFront [10]. The program uses a webcam or image to determine the HSV threshold of specific colors. It makes determining the HSV of a color simple since it implements trackbars and depicts which colors are being excluded as the hue, saturation, and value are manipulated within an image. Using this program, I was able to determine the HSV range of a bright pink post-it, which is the object I used as a selection tool in the demonstration of my project.

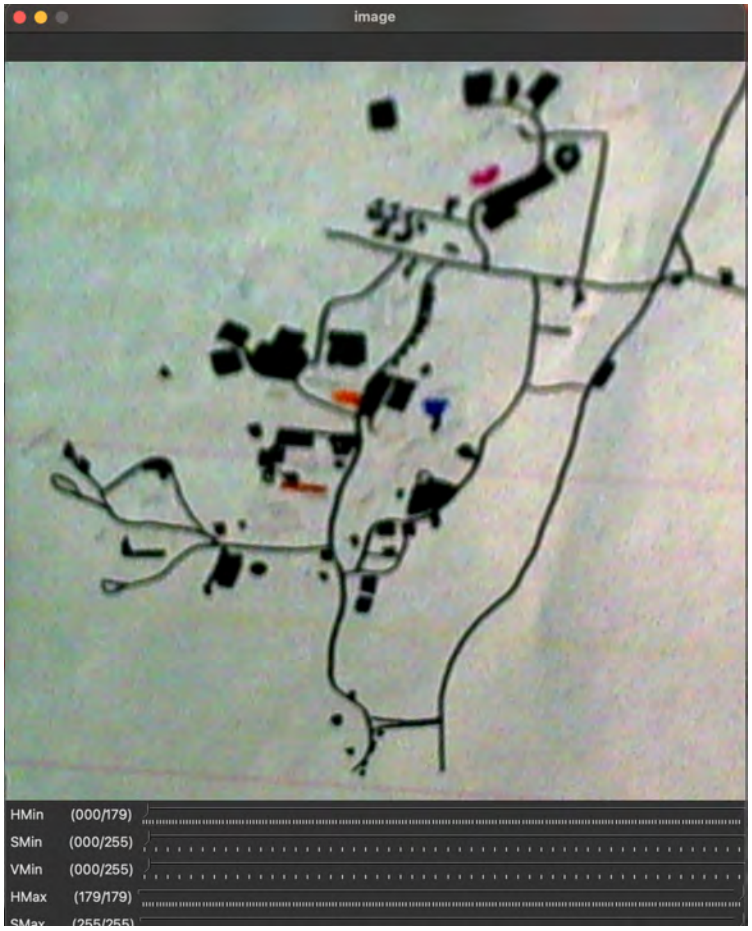


Figure 19. Ponzi's *threshold.py* program applied to a finished, traced map of Bard (Author's Photo, 2022)

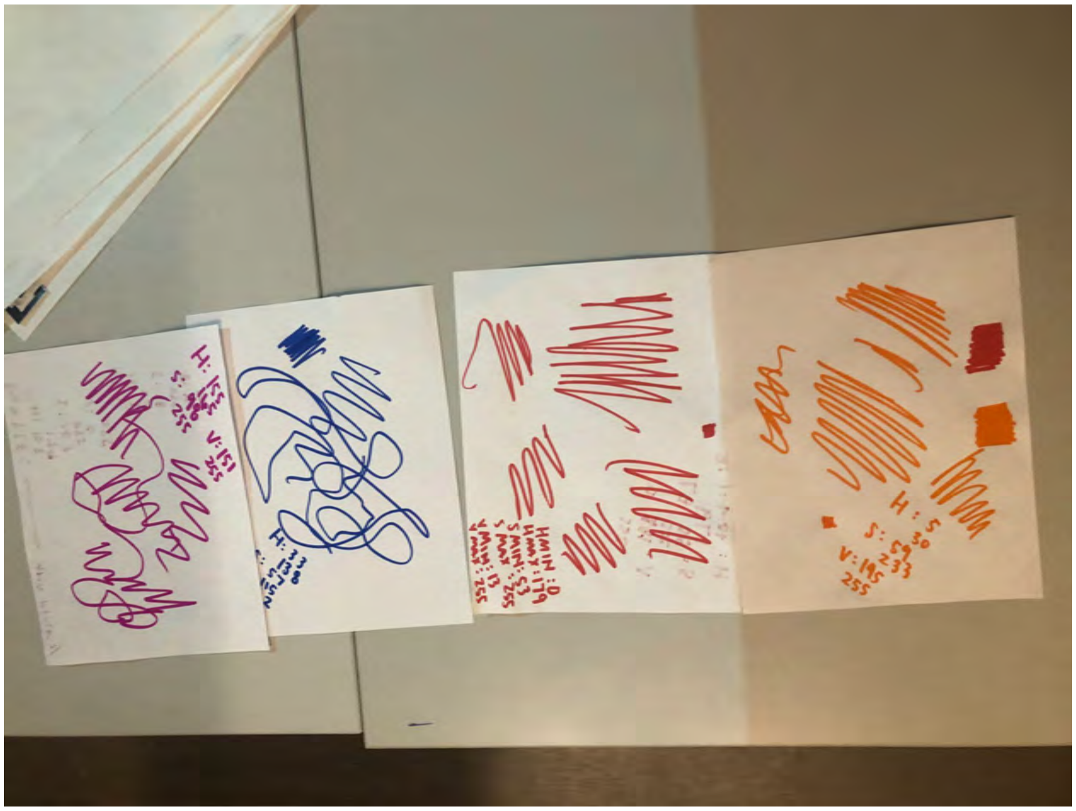


Figure 20. Marker scribbles used to test marker HSV in *threshold.py* (Author's Photo, 2022)

After determining the bounds of the color, in this case pink, that I want to detect, I convert the webcam footage from BGR to HSV and create a mask based on the color's upper and lower bounds within the webcam scope. After the mask is created, I dilate the mask in order to maximize the color detected, and I finally determine the contours of color detected. This color detection is drawn from the "OpenCV and Python Color Detection" article written by Adrian Rosebrock [1]. With contours detected, I use a for loop to determine which contours have an area that exceeds the threshold I set. If that threshold is exceeded, then the colored object is considered to be detected, and its intersections can be tested.

Selection

To determine whether my selection object overlaps with the segmented portions of my map, I created an *intersect.py* program. The method takes in the top-left and bottom-right corners of the segmented map coordinates as well as the top-left and bottom-right coordinates of my post-it contours. Based on the rules of intersection, the two rectangles drawn from the top left and bottom right corners are considered to be overlapping if they share common points.

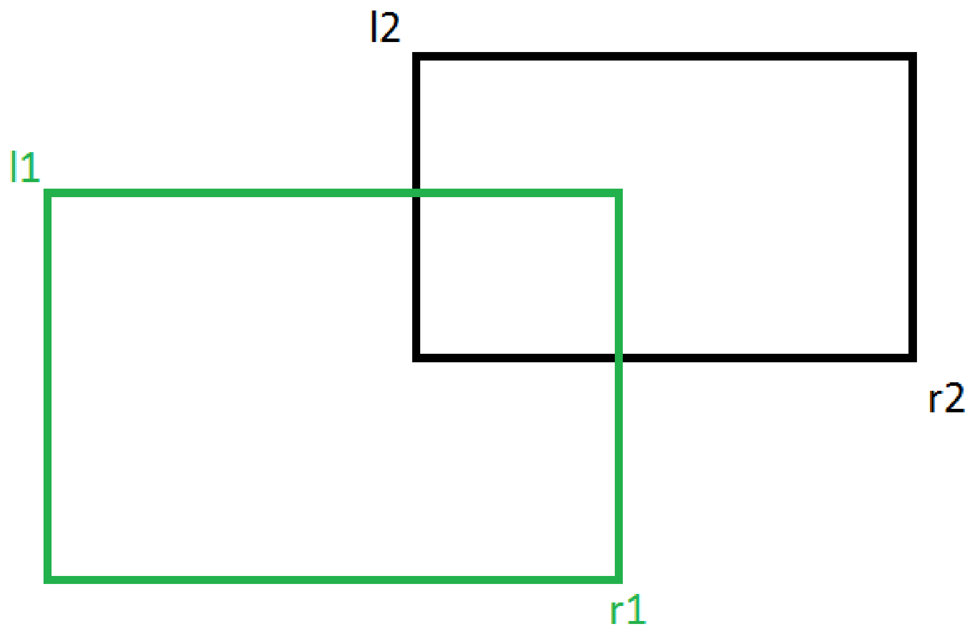


Figure 21. Visualization of rectangular intersections [15]

My *intersect.py* program checks to see if the two rectangles are not overlapping, and otherwise returns True. They are considered to not be overlapping if the *y* value of *r1* is higher than the *y* value of *l2* or if the *x* value of *r1* is farther left than the *x* value of *l2* or if the *y* value of *l1* is lower than the *y* value of *r2* or if the *x* value of *l1* is farther right than the *x* value of *r2*. If none of these conditionals are met, then the method returns True and an intersection is determined. If an intersection is in fact determined, then the ArUco B image projection changes to show an image of the location selected.

Trace Mode

The final portion of my project methods deals with the tracing of the map. The project is meant to result in a traced map on ArUco A where the map is projected. Users can collaboratively trace the map as long as their shadows do not interfere with the light being projected and the ArUco markers on the corners are not covered. The map is meant to be colored in all black, but four colors are used to denote the user's campus preferences. For example, users have the ability to color locations red to indicate their favorite place on campus to socialize. Orange denotes the prettiest view on campus, blue is the best place to study, and purple is the best dorm, or best place to sleep. My *traceMode.py* program uses a screenshot of the drawn map to print this data based on the colors and where they are used. Users are meant to begin by exploring the space using both ArUco windows. This will allow them to determine the names of each building. After writing down the names of the buildings and tracing the outlines of the map in black, the colored markers can be utilized to convey preferences. Once the map has been completed, the pink post-it (or other selection tool) is placed in the small rectangle at location (0,0) on the *mapTransform* table projection. This turns the map projection off for as long as the box is considered to be selected, and screenshots are taken of the map.

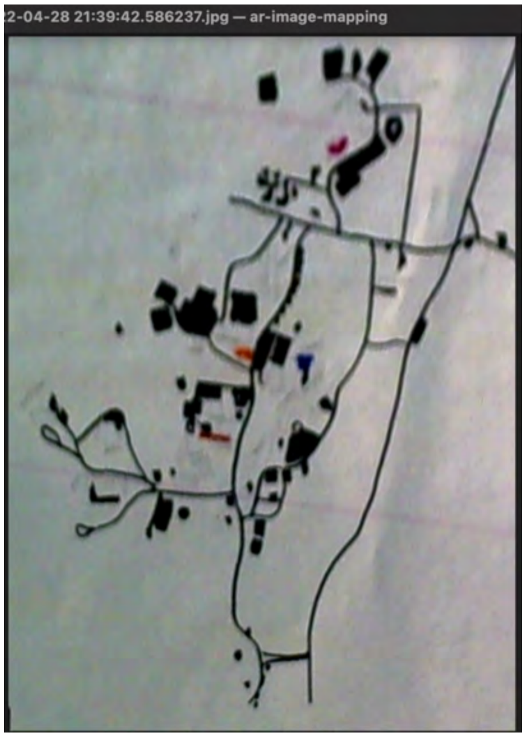


Figure 22. Example of a screenshot taken by *traceMode.py*; note that the image is saved based on the date and time the screenshot was taken (Author's Image, 2022)

The projection must be turned off so that projected colors do not interfere with the color detection of the actual markers used. The screenshots are saved and accessed in the *traceMode*.py program. The red, orange, blue, and purple bounds are determined using Federico Ponzi's *threshold.py* program once again [10]. When those colors are detected in the screenshot at the coordinates determined by the map sectioning, words are printed that detail what the location of those colors means. For example, if the coordinates of Robbins intersect with the found contours of the red mask, then the *traceMode* program will project that "The best place to hang out is Robbins" onto ArUco B when it is detected alone.

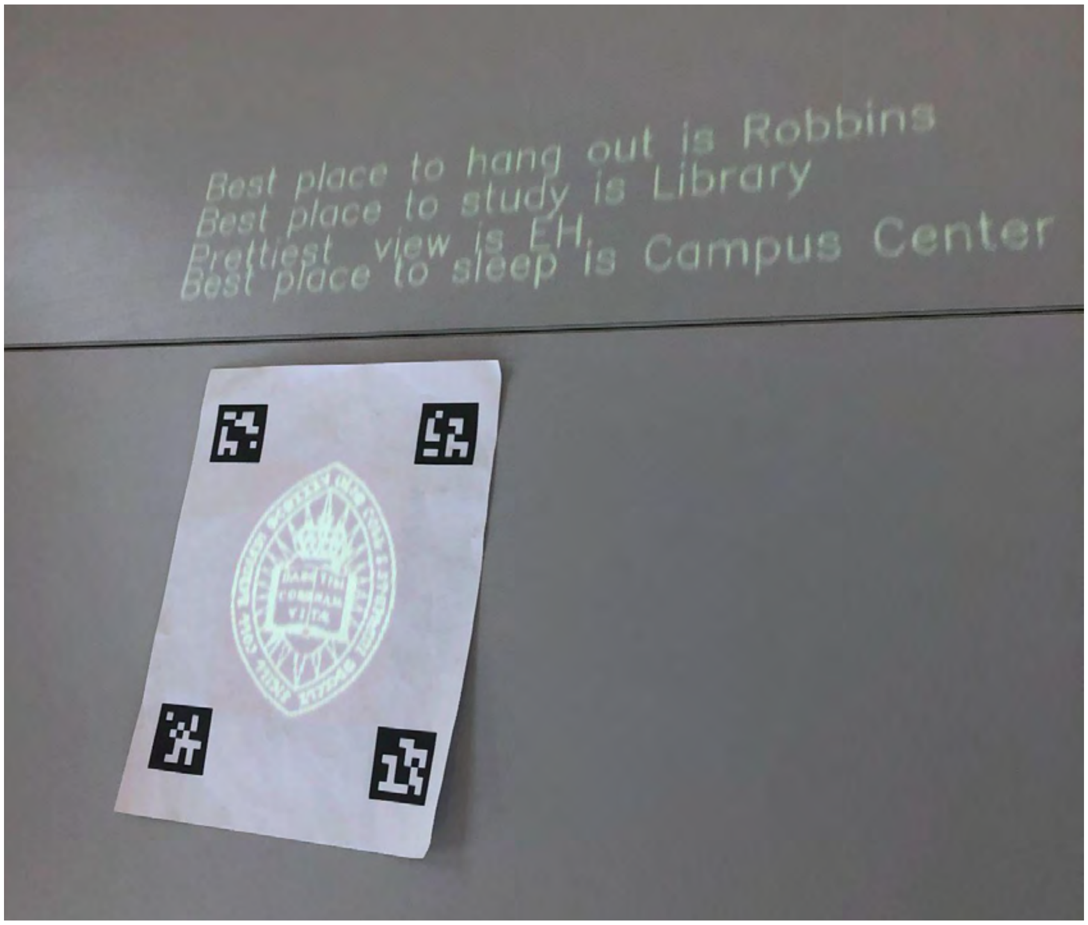

Best place to hang out is Robbins
Best place to study is Library
Prettiest view is EH
Best place to sleep is Campus Center

Figure 23. After *traceMode.py* is executed, the results of the detected colors are projected when the second ArUco screen is placed in the frame.(Author's Photo, 2022)

This piece of the project is meant to incorporate an aspect of subjectivity into the process of exploring the space. Users can input their own opinions about the campus based on the colors they utilize and those opinions, ideally, will be stored in a way that future users can see.

Evaluations

Paper Detection

One vital component of this augmented reality project is paper detection. Pieces of white paper are used in the project to act as moveable windows where images are projected. Since they are designed to be moveable, a homography matrix must be determined for each paper so that the proper image can be depicted with the correct transformations on each piece. This must be constantly updated so that when the paper is moved in the camera scope, the projection also moves to map properly with as little delay as possible. If the homography matrix between the paper and the projected image is incorrect, it can lead to a warped projection, which can cause errors in the AR experience. For example, if users are collaboratively tracing the Bard map, the paper is bound to shift. Proper paper detection will update the map projection to shift with the moving paper to capture the most optimal environment for tracing.

To achieve this perfect mapping, I tried two methods of paper detection. The first involved feature detection. Created by D. Lowe, the Scale Invariant Feature Transformation (SIFT) algorithm is an effective way to determine key points and computed descriptors in an image [14]. It is particularly useful in determining matching features between two images because it is “not affected by the size or orientation of the image” [2]. The algorithm relies on Difference of Gaussians (DoG) which is another algorithm that blurs and subsamples the image it is applied to in an effort to determine ‘blobs’ based on changes in the image scale. After blurring is completed using the DoG algorithm, the image is “searched for local extrema over scale and space" [14]. The key points themselves are computed by comparing points in various scales of the Gaussian Pyramid computed on the image by the DoG algorithm. If a point at one scale is considered to be a local extrema compared to points in neighboring scales, then that keypoint is considered to be best represented in that particular scale [14]. Key points must exist within the threshold of the algorithm, and if they do not meet that threshold, they are rejected. This component of the algorithm "eliminates any low-contrast key points and edge key points and what remains is strong interest points" [14]. Orientation is assigned to these key points to apply image rotation invariance, and descriptors are computed from the 16x16 pixel neighborhood the key point is drawn from. This descriptor neighborhood is used to achieve more precise matching based on the key point. Finally, the actual key point matching can occur by identifying the "nearest neighbor," but since the second closest neighbor can also be a close match, there is a threshold to determine how close the two can be in order to be considered a match. According to D. Lowe, the algorithm "eliminates around 90% of false matches while discards only 5% correct matches" [14]. Its effectiveness was the main reason why I settled on the SIFT algorithm for my feature detection, rather than relying on a Brute Force OpenCV algorithm.

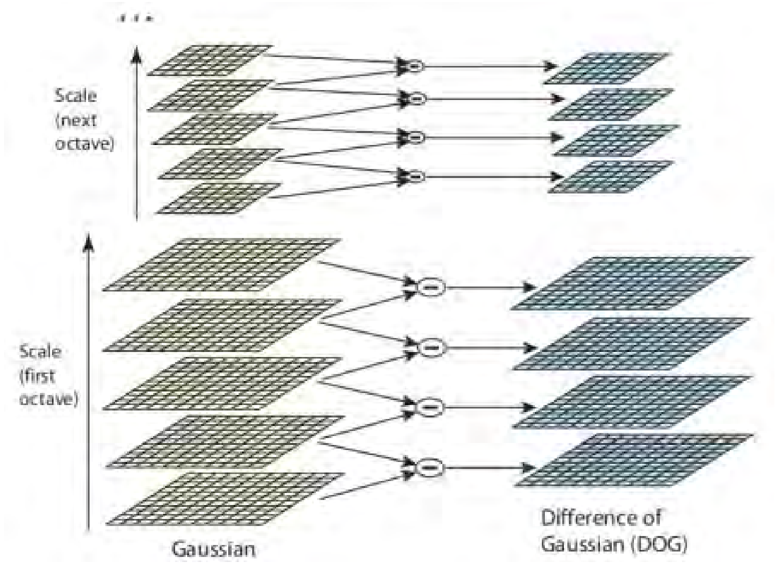


Figure 24. Sift algorithm depiction of Gaussian filters to determine keypoints [14]

The key points and descriptors from the algorithm are computed after the *sift* object is created using *cv2.SIFT_create*. The *detectAndCompute* method returns the key points and descriptors from an input image. The Fast Library for Approximate Nearest Neighbors (FLANN) contains other algorithms that I used when computing the similarities between two images. The *FlannBasedMatcher* is used to determine which algorithm to use, which in this case is the *FLANN_INDEX_KDTREE* algorithm. The *flann* object created from the parameters is used to find matches between the descriptors computed from the *detectAndCompute* algorithm. The good matches from the computation are used to determine points for computing the homography between the paper and the map projection.

After successfully implementing a feature detection based map projection method, I decided to attempt an ArUco marker based projection method. ArUco markers are designed specifically for pose estimation in augmented reality environments and are explicitly defined in OpenCV for fast detection. The program I wrote finds the ArUcos in the webcam input and based on the bounding box corners detected from the *ArUco.detectMarkers* method, projects the Bard map within the borders of the ArUco markers. This ArUco detection program is also successful in projecting a moveable map of Bard onto the paper presented with ArUco markers in each of the corners.

After successfully implementing both SIFT based tracking and ArUco based tracking, I have found ArUco based tracking to be more efficient and accurate. The SIFT based program requires expensive calls to calculate the key points and descriptors. After these calls are made, an expensive for loop is required to cycle through the matches to determine which ones are “good.” With the moving paper, the key points are constantly moving, changing, and being lost. This leads to a constantly warping image that is inconsistent and unstable. If too many key points in the webcam input are not detected, the projection image will glitch and fail to track properly until the key points are detected again. Though successful under the proper circumstances, this program is not ideal for my augmented reality system, particularly the tracing component of the experience.

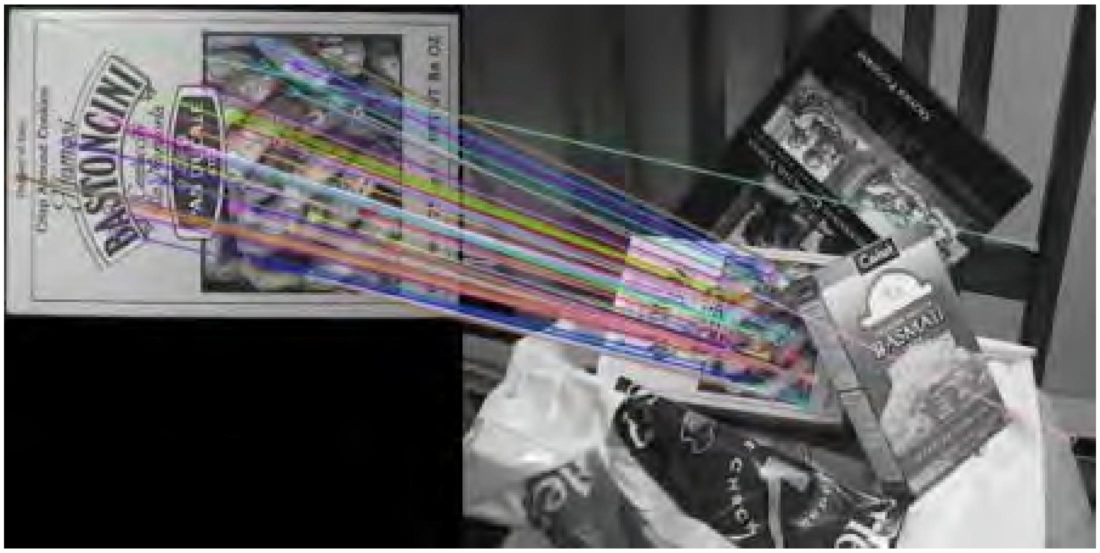


Figure 25. Feature matching example using SIFT [14]

The ArUco tracking, on the other hand, was successful in constantly projecting a crisp map in the center of the ArUco markers. Since the ArUco algorithms are designed to detect those specific objects, they are highly accurate. This program also allows me to relocate the ArUco tags instead of having them printed specifically on an 8.5inx11in piece of paper as the SIFT program required for feature detection. This algorithm is fast and highly accurate. When tracing a map using the SIFT algorithm versus the ArUco markers, it is clear that the ArUco program projects a map perfectly overlaid with the physical drawing despite any changed orientation of the paper. For this reason, I decided to rely on ArUco tracking in my overall final exhibition.

Color Selection

In comparing the use of a laser pointer and a physical object as selection tools, I decided that a physical object was more convenient. Any object can be used if thresholding is done to determine the specific HSV of that object, which makes it a more accessible alternative to laser pointers since laser pointers are not a common tool on hand. Physical objects also allow users to more physically engage with the space. By touching the map itself and by physically moving a tangible object, the physical element of this tour alternative project is maximized. Utilizing a laser pointer also revealed issues. The brightness of the laser pointer can reflect back against the projection and distort what is being selected. Since the laser is so bright, it can be difficult to control the exact bounds that are being detected, and those bounds vary based on how close or far the laser is used from, as well as the lighting conditions of the room. By using a stable object as a selection tool, the size of the tool can remain consistent and predictable regardless of lighting conditions, though the exact color of the object can change depending on how bright or dark the environment is.

Map Sectioning

Originally when determining the sectioning of my map, I determined the bounding rectangles on the original image that I would later project. This made it possible to get specific coordinates from the image without requiring projection in the coordinate determination. After testing, however, the coordinates of the locations that I inputted were slightly off. In an effort to simplify this sectioning, I ran both my *aruco-tracking-perfect.py* method at the same time as my *sectioningMap.py* so that I could click on the coordinates within the live projected map rather than within the image itself. This gave me a more accurate representation of the points that I wanted to use for the top-left and bottom-right rectangles of spaces, since it allowed me to take into account any updated warping and wavering that the map experienced in the location of the ArUco markers.

Initially, I was also utilizing OpenCV's *polylines* function [14]. This gave me the ability to construct polygons precisely around the locations that I wanted users to be able to explore. Though the construction of polylines gave a more precise definition of the spaces to explore, this proved to be a difficult task when it came to finding intersections. With the contours of polylines being less predictable compared to rectangles, it would require more time and effort in the program to determine whether contours of the selection object were intersecting with these specific bounds. This process was made simpler when I switched to defining the sectioned spaces as mere rectangles. It is a less precise way to segment the space, but it allows for less user specificity in the placement of the selection object, since the bounds of the locations are less strict. It also makes my *intersect.py* program incredibly simple since I am simply determining the overlap of two rectangles, rather than the overlap of a rectangle and an irregular polygon. I base my top-left and bottom-right rectangle coordinates on the coordinates determined in my sectioningMap.py polygons, but I depicted the exact rectangles on my *mapProj* image to ensure that there were no overlapping edges between the spaces meant to be explored. Though this is a less accurate representation of the space on the map, it simplifies the project and allows for users to be less specific in the placement of the selection object.

Map Creation and Utilization

I used several different maps in the testing of my project, some of which I made and some of which were pre-existing. The pre-existing maps were useful in determining the spatial relationships of the campus such as where buildings were located in relation to one another, where roads and bodies of water are located, and where parking lots exist. Despite being useful in teaching me how to set up the spatial relationship of the campus itself, I knew from the beginning of this project that the map I wanted users to trace should be as minimalistic and bare as possible. This would require users to explore the space with ArUco B in order to construct their own map without relying on the labels and color indicators of a premade map of the campus.

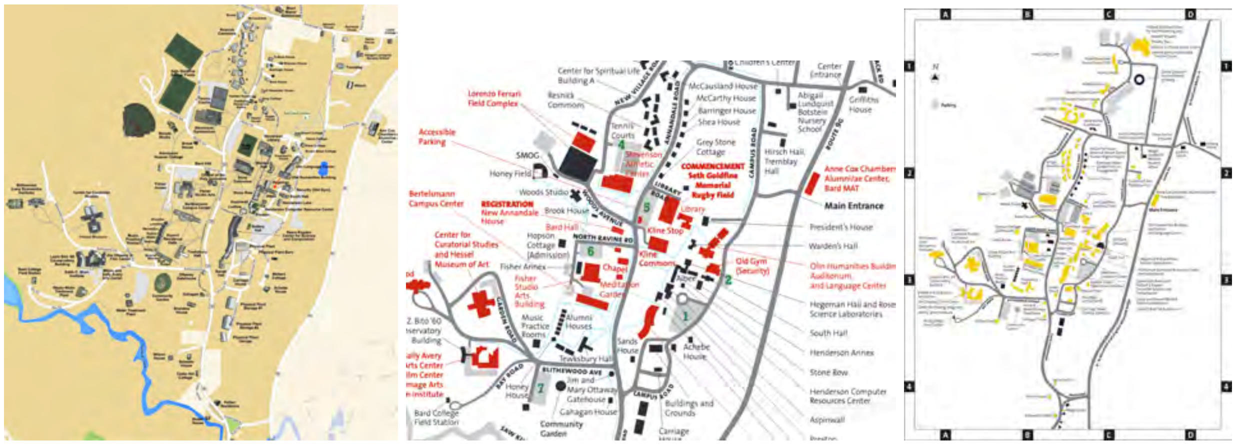


Figure 26. Pre existing maps of Bard that I utilized in initial testing [3][4][8]

To create the minimal map that I envisioned, I utilized the map on the top-right to trace the general layout of the campus. Using the photo editor GIMP, I was able to easily extract the information that I did not want present in my projection map. The colors I used for the exact projection map proved to be incredibly significant. I could not include any colors that could possibly fall within the threshold of the selection object, so in the case of my demonstration, I avoided colors like pink and red in my map. The background color was also significant. I tested using both white and black backgrounds and found that though a white background is useful in making the projection vibrant for users, the brightness of that projection makes camera detection difficult and complicated.

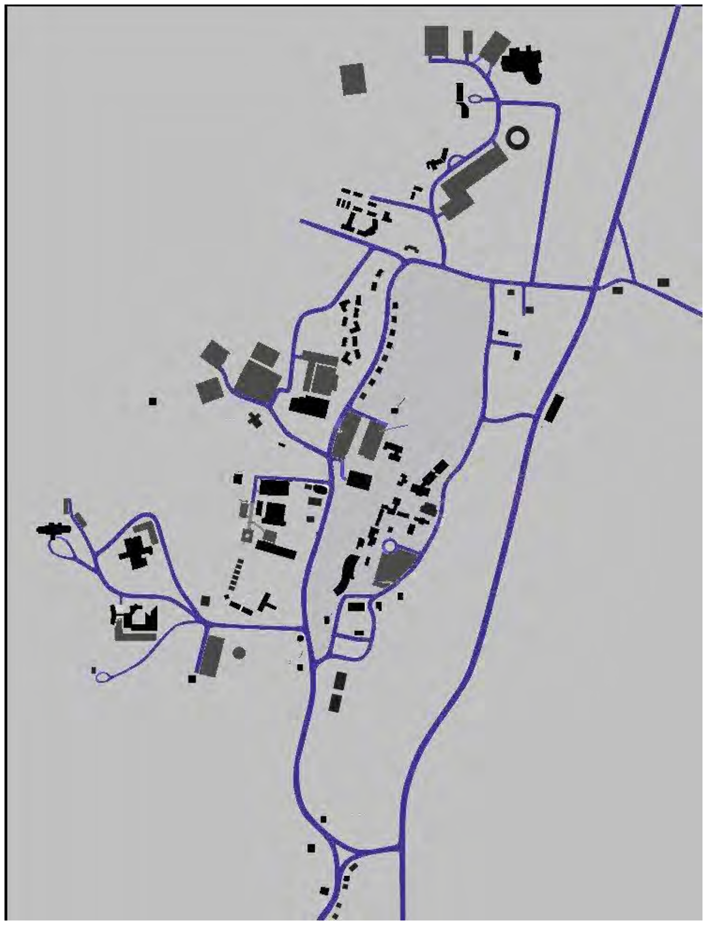


Figure 27. Example of a light background map that, when projected, would disrupt webcam feed (Author's Image, 2022)

In relying on a solely black background, I minimize the extra lighting from the projection that could influence the camera detection. I originally used blue to indicate the roads throughout campus, gray for the parking lots, and white for buildings; however, as my testing continued, these colors changed a bit. There are many small buildings on campus that I found to be unimportant when exploring the campus from an outsider perspective. Buildings like the Manor Gatehouse and the Hannah Arendt center did not seem as necessary to include in the exploration of the campus, so rather than separating my map into roads, parking lots, and buildings, I separated it into roads, unexplorable locations, and explorable places. The roads remained blue, unexplorable places, including parking lots, remained gray, and explorable places are now depicted in a vibrant green that makes it easy for users to see.

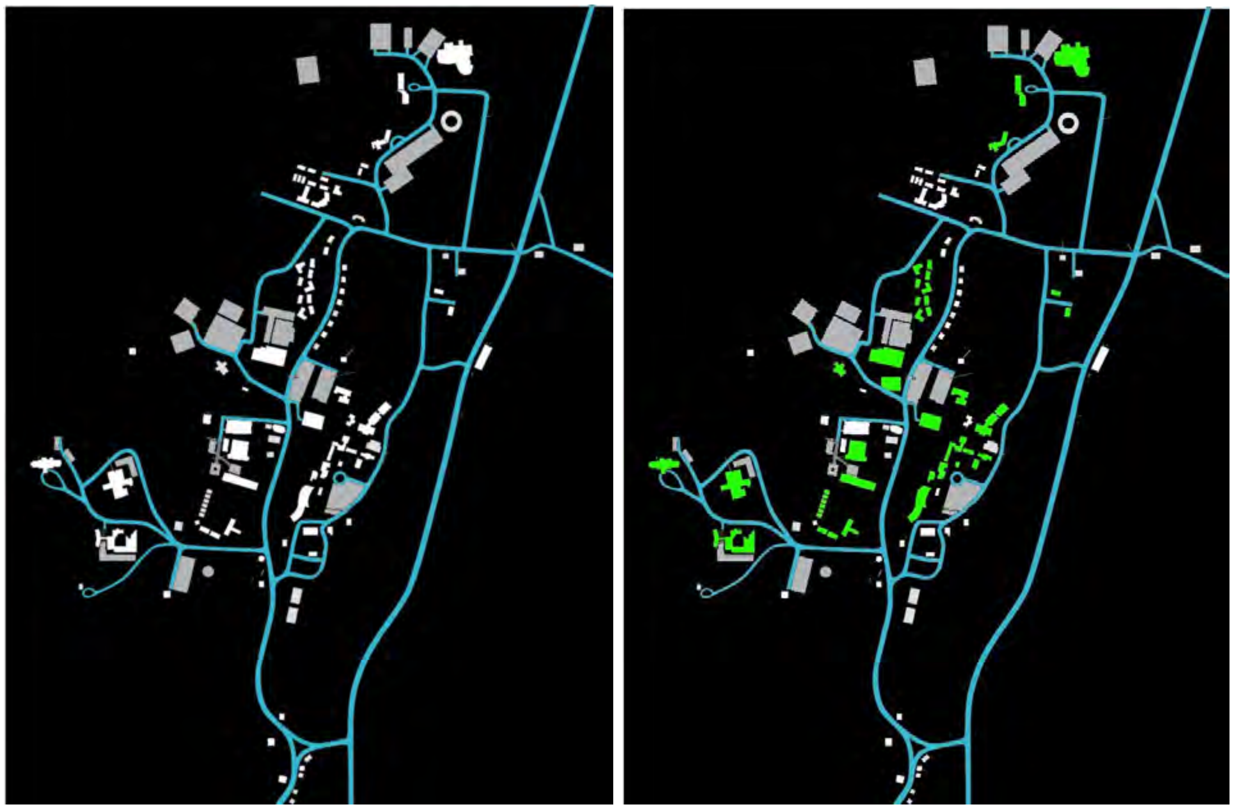


Figure 28. The original black background map (left) and the final black background map with locations emphasized in green (right) (Author's Image, 2022)

This vibrant green draws users to those specific locations rather than wasting their time by placing the selection object on locations that have nothing attached to them. It would be simple to go in and change these locations as well. Parking lots could be included in the map sectioning and the project could include them as places to explore, but I wanted my project to emphasize the most important aspects of campus without cluttering the user's mind with less useful information.

Trace Mode

Attempting to construct the *traceMode.py* functionality of my project proved to be the most inconsistent and sensitive. Originally, my plan was to detect the colors used on the map in real time and as users colored portions of the map, projected words would state what colors in those specific locations indicate. This was an impractical feat for several reasons. The thresholding of colors is far more difficult to complete in a live image since shadows and movement are constantly changing the HSV of the color being detected. Skin tone and other objects introduced into the webcam feed could also be read and detected as falling within the range of colors that should be detected, when in reality, they are not a part of the project. The projection of the map itself also changed the color of the marker ink. For example, if a user colored in Robbins with the purple marker, the color purple detected by the camera would be different from the pen's actual ink. The camera would detect the green of the Robbins map projection combined with the purple of the marker and would result in a new HSV value. When I realized that the projection was distorting the actual HSV of the marker ink, I originally planned to simply run my projection program alongside my thresholding program in an effort to get the specific HSV of the ink and projection color combined. Though this would have been successful, it was unnecessary.

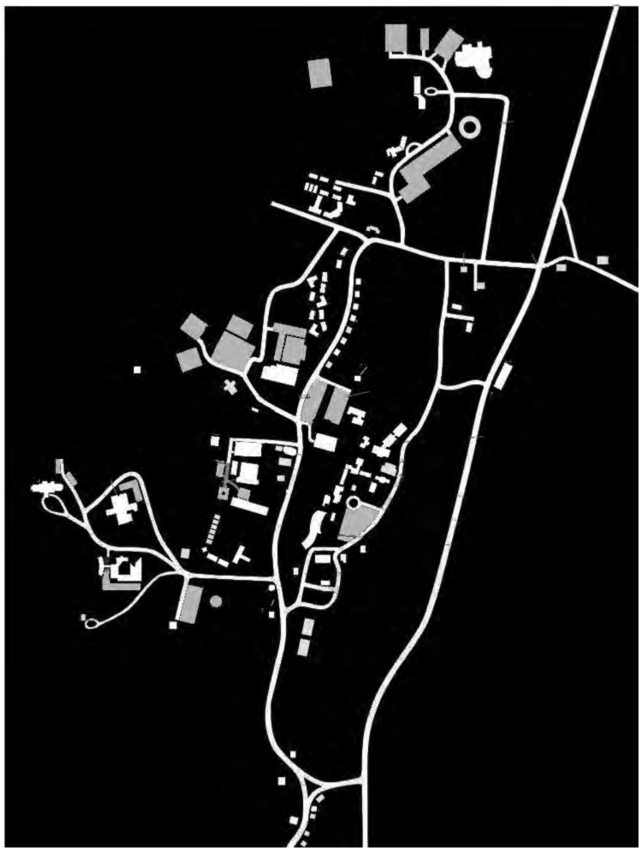


Figure 29. The black and white map used for testing projection color distortion on marker ink.(Author's Image, 2022)

I also considered creating a separate setting entirely for tracing the map. This setting would require much more work on the user's side. They would begin by exploring the space with ArUco B, and would trace the map using a pen, labeling the buildings. Then, by placing the selection tool in a certain location on the projection, the "Trace Mode" would be activated in which the projected image would change from the neon green map to a solely black and white map. This would prevent the neon green of the original projection from disrupting the color of the marker ink. Though this was an alternative solution to my color thresholding problem, it was also unnecessary and complex. The bright white of the projection buildings, though less drastic, still changed the color of the marker ink in the webcam feed. This finally led me to simply momentarily pause my projection so that the marker ink could be properly detected.By including a small box that works as a screenshot button in the top left corner of my projection, I allow users to temporarily pause the projection so a screenshot can be taken. These screenshots allow the upper and lower bounds of the colors detected to be true to the actual color rather than be reliant on the projection it is layered with. They also provide a digital archive of past tours' map tracings with the projection excluded, which is useful in tracking the progress of this project's usage. Once the tracking object is removed from the screenshot box, the projection will continue. This allows for multiple screenshots to be taken over the course of an instance of this project.

Demonstration

The demonstration of this project took place on April 9, 2022 at the Experimental Humanities' New Annandale House for Admitted Students Day. I arrived at 10am to mount the projector-camera system and test the project before tours began. The weather was overcast, so I adjusted the webcam settings to most accurately detect colors and contours with the given lighting conditions. I gave a short introductory presentation for the project in which I summarized my abstract and communicated the importance of the project. The introduction was followed by a demonstration of how the project worked. I began by calibrating the system using the OpenCV chessboard image. After calibration was completed, I placed the large ArUco paper in the camera frame where the map was then projected. I then placed the smaller ArUco paper in the frame where the logo was projected. The projections depicted minor flashing due to inconsistent detection of the ArUco markers, but were generally consistent.

After placing my ArUco screens onto the tables, I then moved the pink post-it across the map to show how buildings were augmented using the smaller paper. When I moved the post-it over the neon green projected buildings, the second paper depicted the name and image of that location. I ended the presentation by talking about how users were meant to trace the map, and showed how a screenshot can be taken by placing the post-it in the top left rectangle of the projection.

My project was being showcased at this event as an example of how Experimental Humanities projects are interdisciplinary. Since my project focuses on architecture and visual art in conjunction with computer science, it was an accurate representation of how these disciplines are interconnected. In terms of the success of the project demonstration, my program ran with no errors and was successful in projecting an interactive map of the campus. My code worked as expected, and received positive feedback from prospective students and their families. Though I would have considered it to be a more successful demonstration if others had the opportunity to engage with the map themselves, the large number of people and the short time constrictions made this impossible. The demonstration was nevertheless a useful exercise in verbally communicating my project overview to a non technical audience. It provided me the opportunity to answer questions about how my system works on a broad scale, and allowed me to present what the project is capable of.

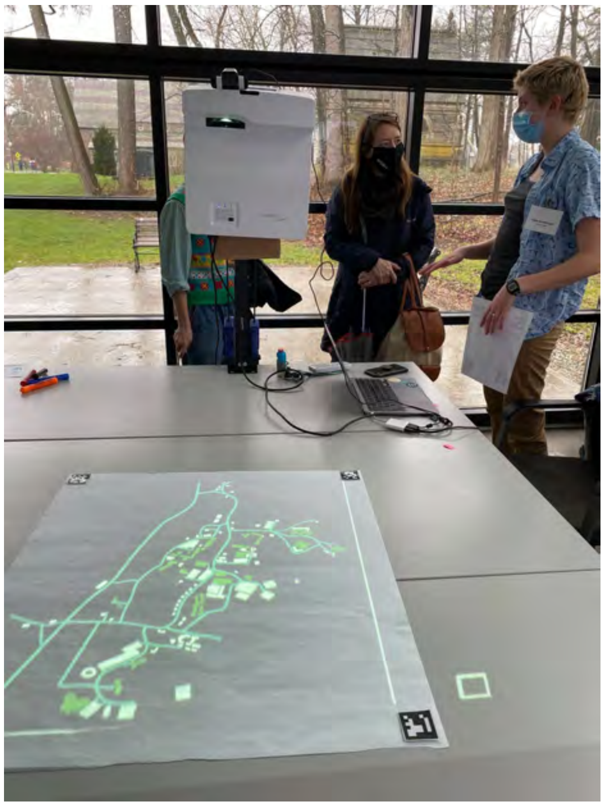


Figure 30. Pictures from the EH demo (Author's Image, 2022)

Conclusion

Overall, I was successful in constructing an augmented reality environment that explores Bard's campus using a projector-camera system. My project presents an alternative way of physically exploring space that is accessible to users who struggle with mobility. Users are able to uncover parts of campus quickly without having to walk across 1,000 acres, and they have the opportunity to leave their subjective touring experience for future users to see. By using this system, users are more involved in their tour experience rather than being confined to the strict route tour guides follow. Users also have the ability to see all parts of campus, not just the select few included on general campus tours. The user involvement and explorative aspects of my project make it a more memorable and meaningful experience, especially since other colleges are not providing prospective students with the opportunity to engage with their campuses in this way. This is an example of a computer science project that is designed for non technical people to experience and enjoy. It allows users to be in control of their experience and encourages them to be creative and collaborative throughout their tour. All of these qualities coalesce to form an interdisciplinary experience that can be easily categorized as an Experimental Humanities project.

Limitations

This project, though successful in many aspects, presented me with several challenges that limited its overall effectiveness. Camera-projector systems in general are known to be reliant on the environments where they are installed. Light conditions greatly affect the efficacy of my project, particularly when it comes to detection functionality. Since my color detection is based around color thresholding, the HSV value thresholds of colors are specific to the lighting conditions of the space when the thresholds were determined. For example, the pink post-it note could appear within the camera frame with HSV values during the daytime, but those HSV values are much different when the thresholding is applied at night. Cloudy versus sunny conditions also play a considerable role in the thresholding of colors.

Color detection is not the only detection that is reliant on lighting conditions. ArUco marker detection also varies in efficacy depending on the lighting. If the environment where the project is being presented is not well lit, the contrast between ArUco markers and the paper they lie on is less accurate. When ArUco markers are less obvious within the webcam frame, the projection is less accurate and can lead to an inaccurate tracing of the campus map. This in turn could lead to inaccurate information collection regarding the most popular places on campus.

If my project were permanently installed in a room with consistent lighting conditions, these limitations could be easily overcome. With software like the Mac Webcam Settings application, it is possible to set specific camera parameters like camera focus, contrast, saturation, and sharpness. These specified settings make it simple to create a webcam feed that consistently detects colors and borders based on the stable lighting conditions; however, since my camera-projector system is meant to be portable, these settings cannot be preset. Instead, during the execution of the project, the user must adjust the camera settings to maximize the accuracy of camera detection. Even with these settings being adjusted, as weather conditions and sun position change constantly, the project must constantly be adjusted to account for these changes. If a more permanent installation of this project with less natural sunlight were an option, these limitations could be overcome.

I also faced difficulties with the specific placement of the camera. The webcam I used was placed atop the projector, which gave the webcam feed a slanting perspective. This was corrected using the homography matrices I calculated, as seen in Figure 14; however, since the camera focus is consistent across the plane it is detecting, a slanting viewpoint changes the sharpness of the image. The camera focus was precise for the ArUco markers that were closest to the camera, but the two markers that appear farther away from the camera are less clear due to their distance.

This limitation can be overcome with a better camera installation position. Placing the camera directly above and perpendicular to the projection will allow the projection to be viewed on a plane with a consistent focus. No physical or projected aspects of the project would be closer or farther from the camera, which would make the entire feed cleared after warping occurred. Overall, the limitations of this project would be remedied by a more stable and consistent installation, since lighting conditions could be controlled and the camera focus could be fixed.

Future Work

Upon the completion of this project, there are still steps that can be taken to improve the impact and efficacy of its application. As I mentioned previously in the *Limitations* section, a permanent installation is the first step to making the project easy and accurate in its use. By installing the system in a building with controlled lighting like the Admissions Office, users will have less set up to maintain before executing the program. By being placed in the Admissions Office in particular, prospective students and their families will have the opportunity to explore the campus both physically and virtually. It could also work as an introduction to what the Experimental Humanities program does. Since it is an obscure concentration at Bard, I want my project to make students aware of the opportunities available within the program.

When it comes to improving the abilities of the project in general, I believe this project can be expanded in so many ways. In the future, it would be useful to project pie charts that detail the colored sections detected from tour maps. For example, a pie chart could be projected that, based on the archive of map screenshots from the program, depicts which buildings are considered to be the best places to study. This information would act as a visual representation of past and present user preferences, and would put the subjective color choices during map tracing into a broader perspective.

The project would also benefit from more subjective capabilities. I would like for students, particularly tour guides, to be able to incorporate their own drawings and notes into the map. For example, rather than just projecting images of the buildings on ArUco B, another screen could act as a slideshow of drawings or short notes related to the building selected. Students could draw and write on ArUco screens, then take a screenshot using the pink post-it. After, those screenshots could be added to the list of images in a slideshow that gets projected on a future ArUco screen that is placed next to ArUco B. This would add another layer of subjectivity to the augmented reality tour, and would allow students and visitors to better connect with the space.

Finally, I would have liked to include a more historical component to this project. I would have liked the ability to include a slider that changes the depiction of the map over time. This would show the timeline of this land charted from before Bard was founded, and would both pay respect both to the indigenous people that resided here originally and would acknowledge the path Bard took in its creation. The project is currently successful in presenting an augmented reality view of the campus, allowing user subjectivity, and creating an archive of the finished tours, but future work would improve upon the work already done. It would provide a more accurate view of the school and would give users a more vivid depiction of the student population based on their subjective inclusions.

Bibliography

[1] Adrian Rosebrock. 2014. OpenCV and Python Color Detection. (August 2014). Retrieved from https://pyimagesearch.com/2014/08/04/opencv-python-color-detection/

[2] Aishwarya Singh. 2019. A Detailed Guide to the Powerful SIFT Technique for Image Matching (with Python code). (October 2019). Retrieved from https://www.analyticsvidhya.com/blog/2019/10/detailed-guide-powerful-sift-technique-image-matching-python/

[3] Annandale Online. Retrieved from https://www.annandaleonline.org/s/990/images/editor_documents/Commencement/2020_commencement/comm_weekend_map_fv.pdf

[4] Bard College. Retrieved from https://www.bard.edu/campus/maps/pdfs/campusmap.pdf

[5] Charles Patel. 2020. Underneath Numpy array & Python list. (July 2020). Retrieved from https://medium.datadriveninvestor.com/underneath-numpy-array-python-list-42a30e62f693

[6] Cullen C. Drissell. 2020. A Mixed Reality System for Learning Data Structures. Senior Projects Spring 2020. 302.

[7] Daniel Moreno and Gabriel Taubin. 2012. Simple, Accurate, and Robust Projector-Camera Calibration. In Proceedings of the 2012 Second International Conference on 3D Imaging, Modeling, Processing, Visualization & Transmission (3DIMPVT '12). IEEE Computer Society, USA, 464–471. DOI:https://doi.org/10.1109/3DIMPVT.2012.77

[8] Digital Spaces Unconference. Retrieved from https://hawksites.newpaltz.edu/digitalspaces/location/directionstobard/

[9] Dynamicland. Retrieved from https://dynamicland.org/#mission

[10] Frederico Ponzi. 2021. hsvThreshold.py. Retrieved from https://gist.github.com/FedericoPonzi/1728542ae0c8057e43658a3216e385c5

[11] Geeks for Geeks. 2021. Find if two rectangles overlap. (October 2021). Retrieved from https://www.geeksforgeeks.org/find-two-rectangles-overlap/

[12] Julia Chatain. 2015. SyMAPse: Augmented Interactive Maps for Subjective Expression. Human-Computer Interaction [cs.HC].. hal-01191429v2

[13] Markus Funk, Oliver Korn, and Albrecht Schmidt. 2014. An augmented workplace for enabling user-defined tangibles. In CHI '14 Extended Abstracts on Human Factors in Computing Systems (CHI EA '14). Association for Computing Machinery, New York, NY, USA, 1285–1290. DOI:https://doi.org/10.1145/2559206.2581142

[14] OpenCV. 2015. Open Source Computer Vision Library.

[15] Rahul Sukthankar, Robert G. Stockton, Matthew D. MullinSmarter Presentations: Exploiting Homography in Camera-Projector Systems. In ICCV 2001 (pp. 247-253).

[16] Yahya Ghazwani and Shamus Smith. 2020. Interaction in Augmented Reality: Challenges to Enhance User Experience. In Proceedings of the 2020 4th International Conference on Virtual and Augmented Reality Simulations (ICVARS 2020). Association for Computing Machinery, New York, NY, USA, 39–44. DOI:https://doi.org/10.1145/3385378.3385384

[17] Yalda Shankar. 2021. Homography Estimation. Retrieved from

https://towardsdatascience.com/estimating-a-homography-matrix-522c70ec4b2c

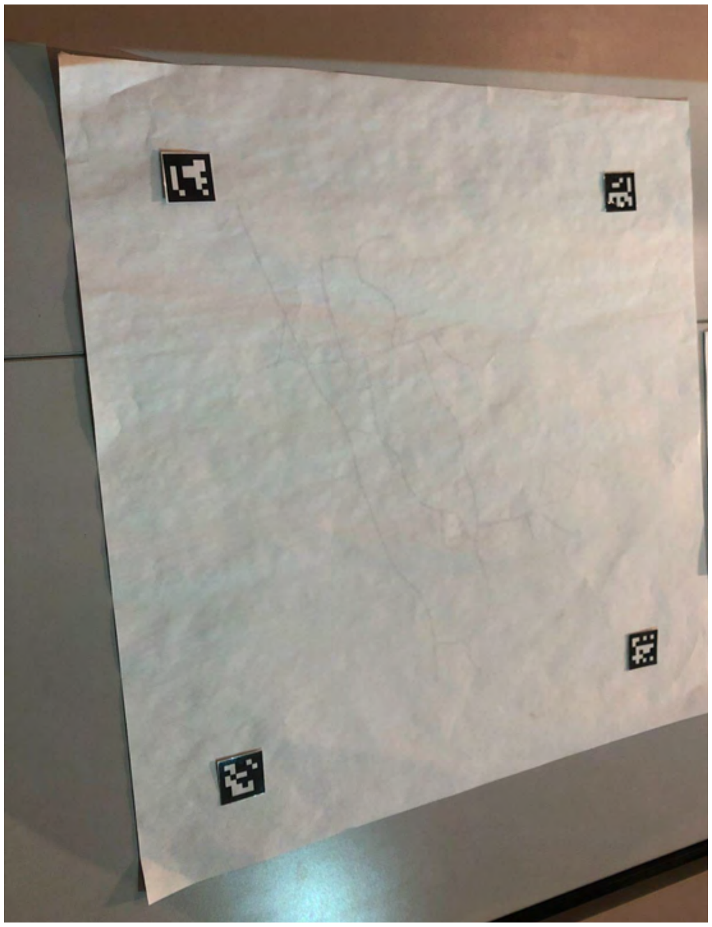
[18] W3 Schools. Numpy Introduction. Retrieved from

https://www.w3schools.com/python/numpy/numpy_intro.asp

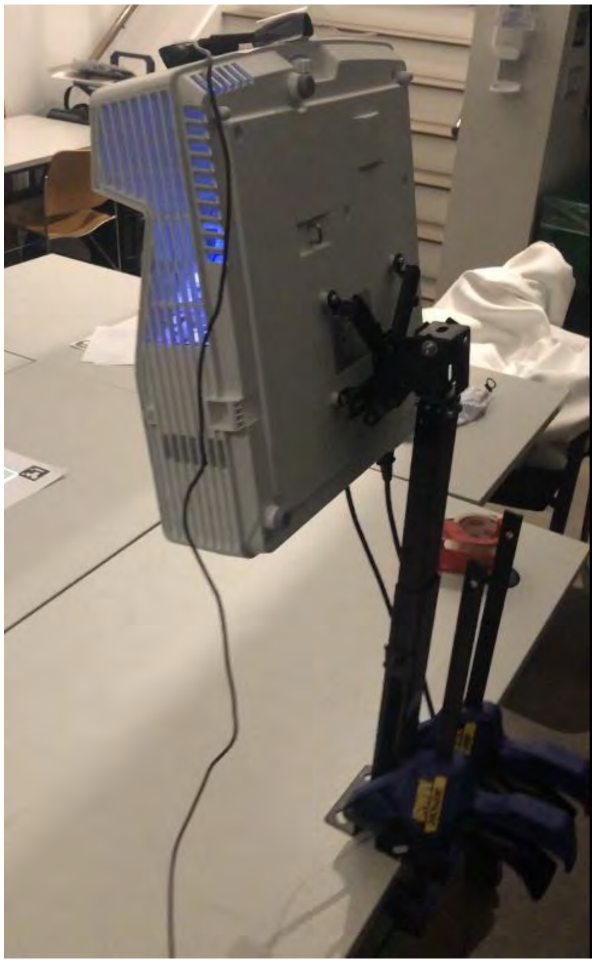
Appendix A: Setup

Step 1. Print out two ArUco tracking papers. One should have ArUco marker 1 in the top left corner, ArUco marker 2 in the top right corner, ArUco marker 3 in the bottom right corner, and ArUco marker 4 in the bottom left corner. The other should have markers 5, 6, 7, and 8 in the same clockwise layout, beginning from the top left corner.

Step 2. Cut out markers 1, 2, 3, and 4 from the printed paper and secure them in the same alignment to a larger piece of paper, creating ArUco A.



Step 3. Attach a mount to the back of the short throw projector using screws and allen wrench. Elongate the neck of the mount to maximize the size of the projection. Using hand clamps, secure the base of the mount, with the projector attached, to the edge of a table.

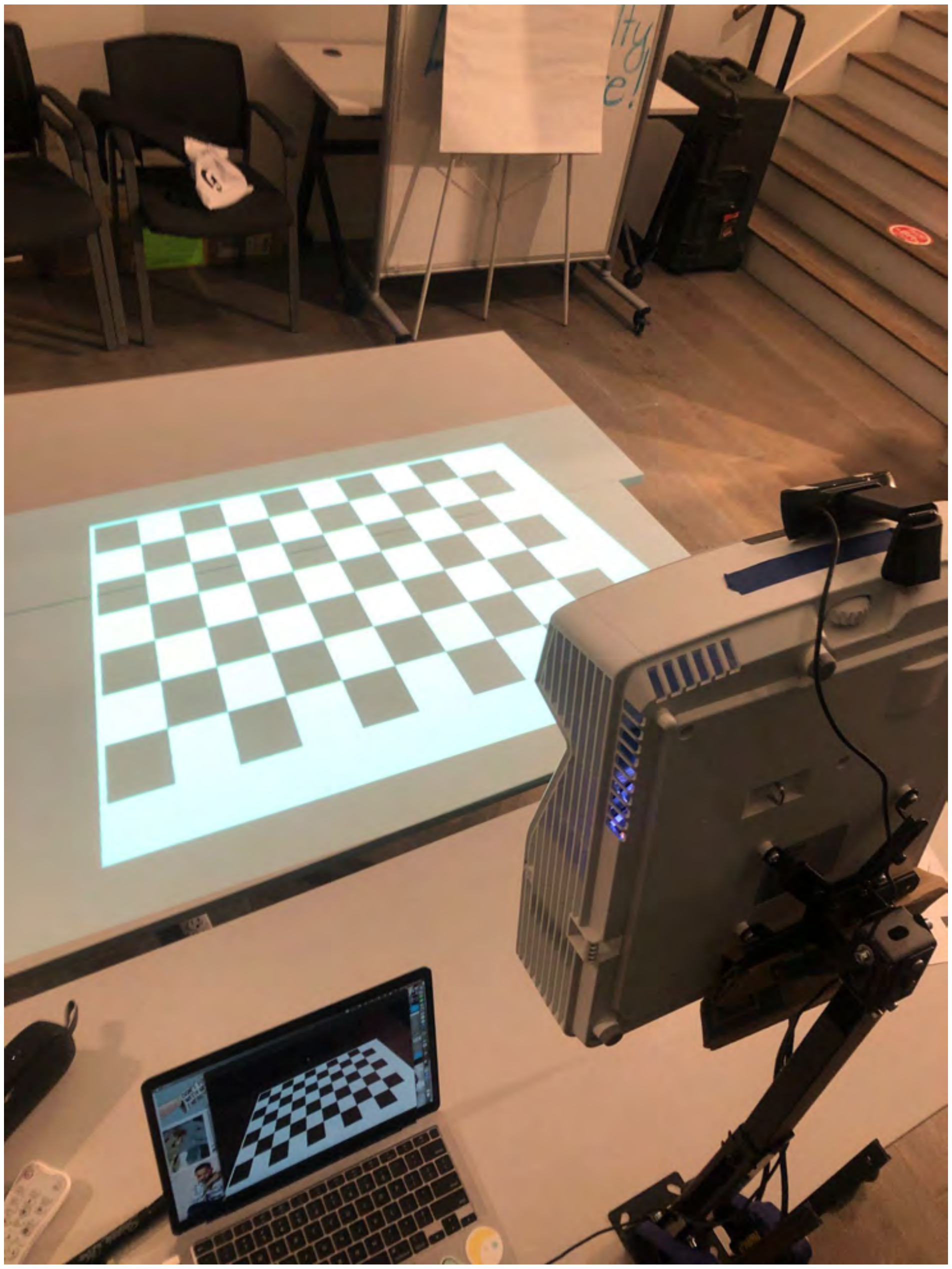


Step 4. Plug in both the power and HDMI chords to the bottom of the projector. Connect the HDMI via an adaptor to the laptop where the code will be executed.

Step 5. Connect the camera via a USB adapter to the laptop where the code will be executed. Set the focus of the camera from auto to manual using Webcam Settings App.

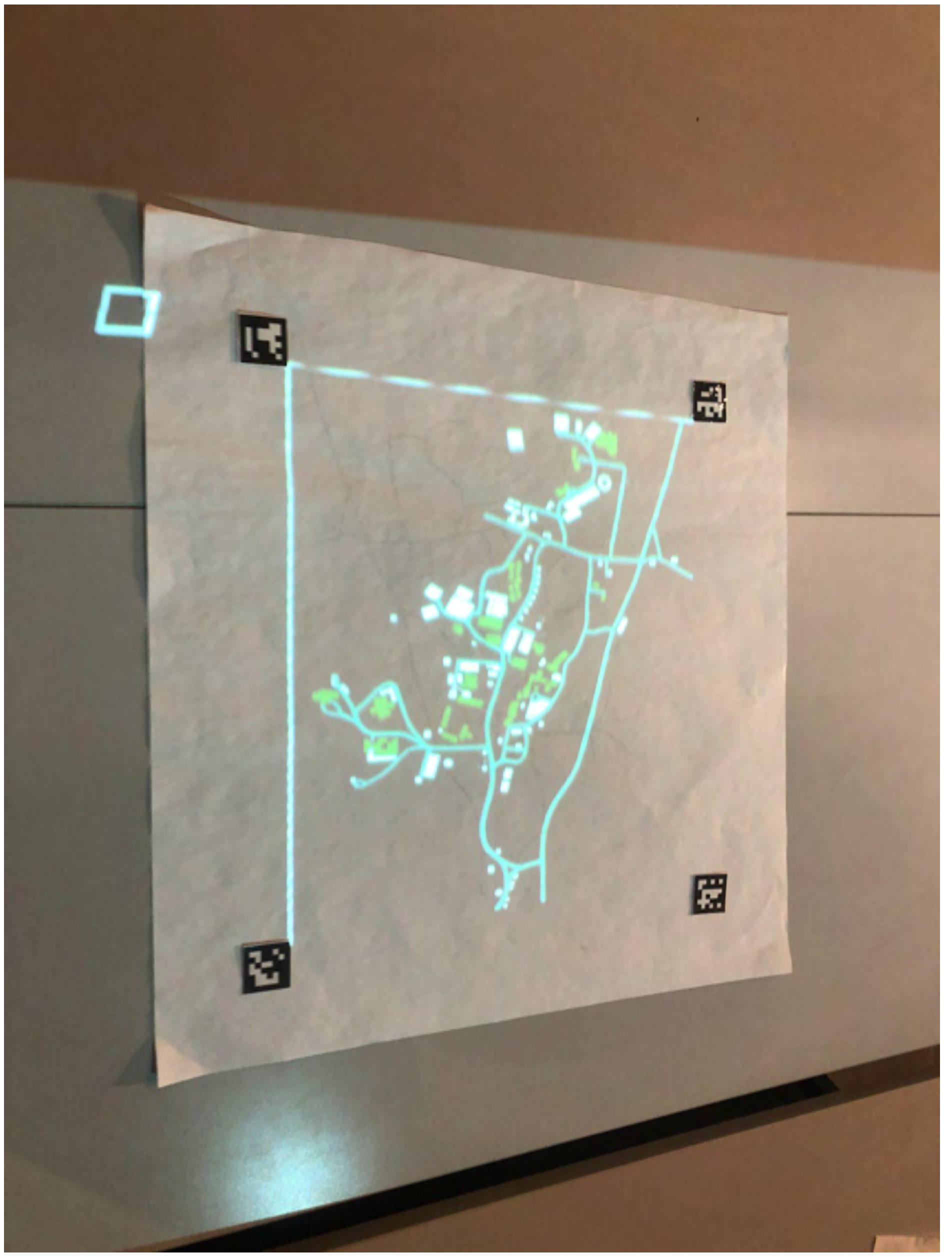
Step 6. Turn on the projector and the camera, and run the *aruco-tracking-perfect.py* program.

Step 7. While the camera is on, make sure the calibration pattern is entirely visible within the frame of the camera.

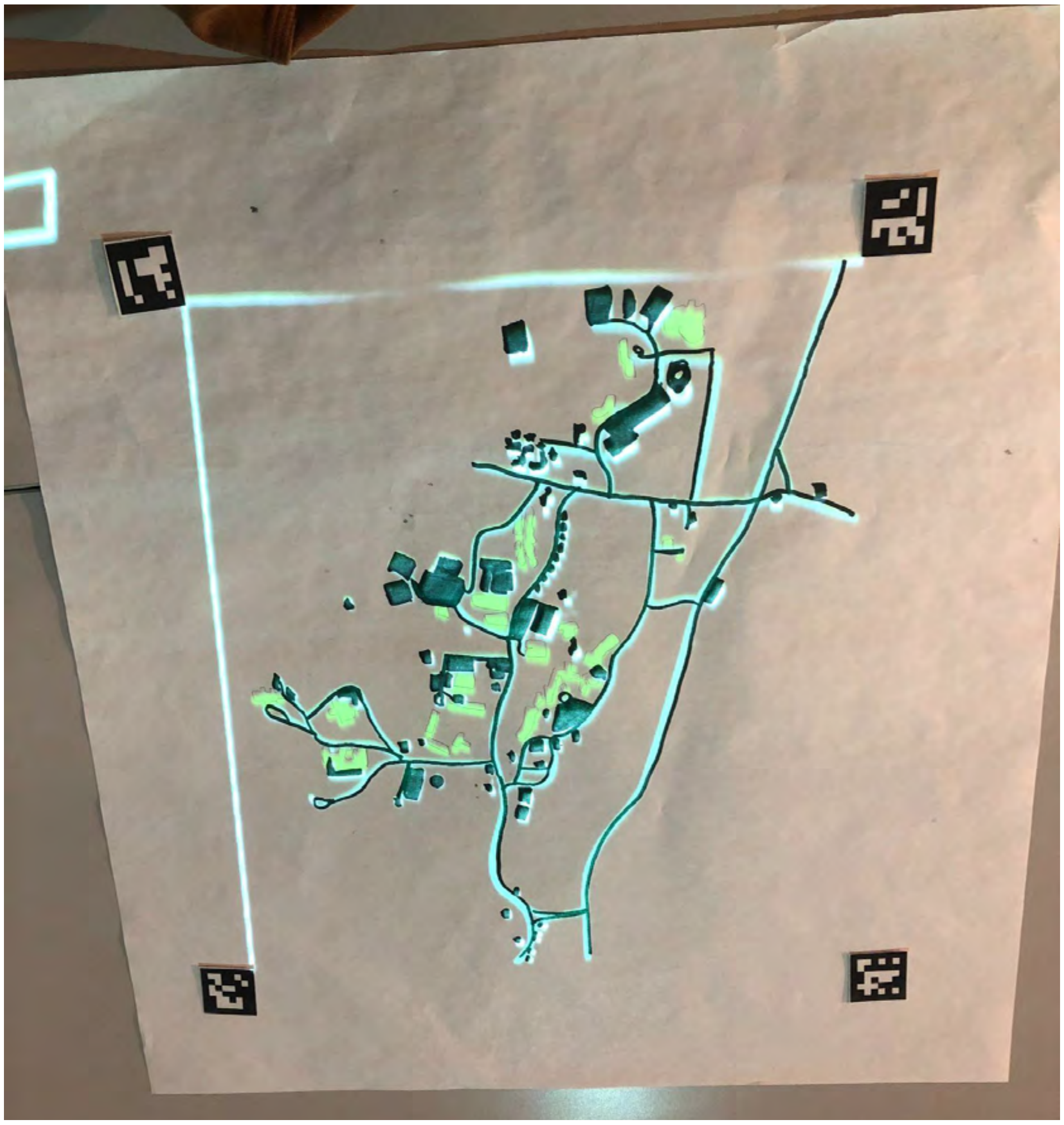


Step 8. Once the projection and camera frames are aligned, press ‘c’ on the laptop keyboard to properly calibrate.

Step 9. Place ArUco A within the camera frame. The basic map of Bard should appear within the marker corners.

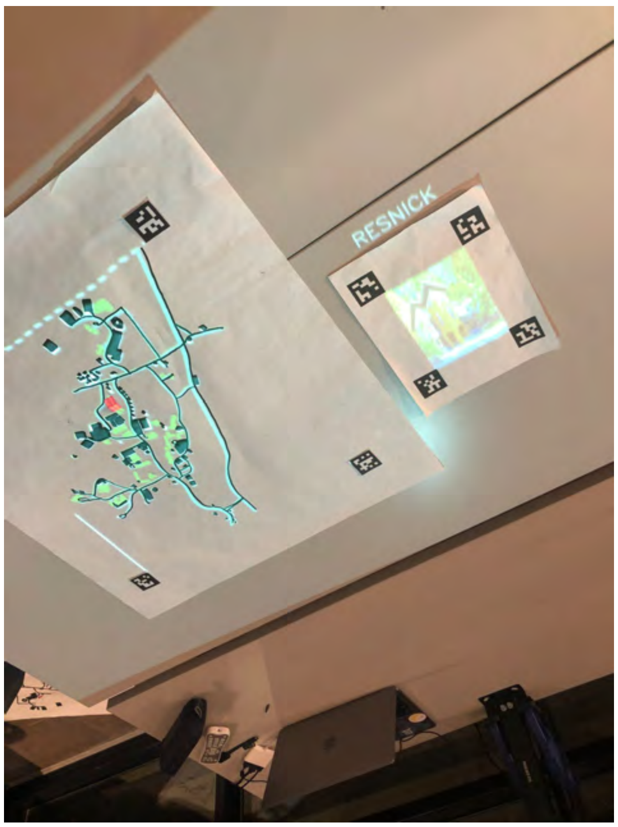


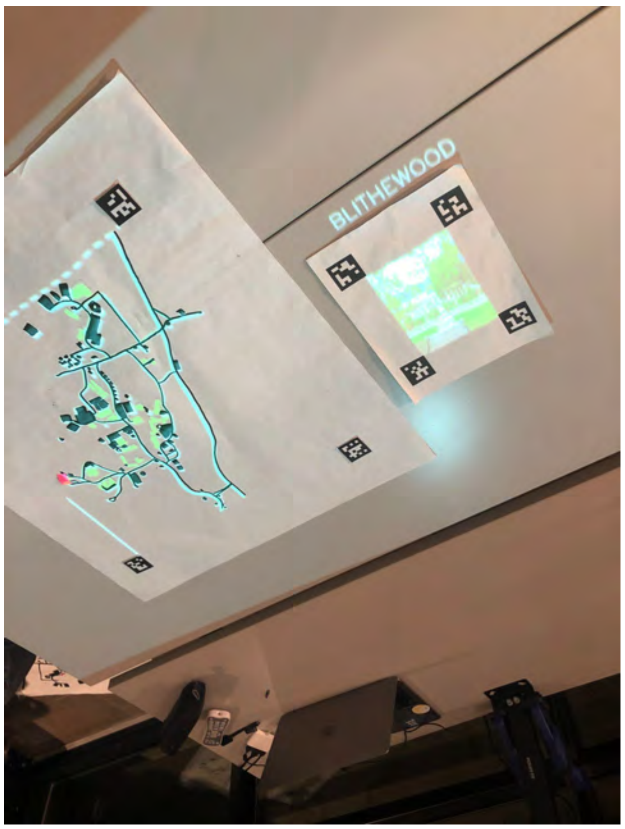
Step 10. Trace the blue roads and color the white boxes with black sharpie. Outline the neon green buildings with a fine pen or pencil.



Step 11. Place ArUco B within the camera frame beside the large paper. If the images depicted on the two papers begin flashing, attempt to alter the camera sharpness, brightness, and contrast to clarify the ArUco markers within the frame.

Step 11. Place a neon pink post-it tab on green locations of the map to explore the space. The image and name of that location should appear on the smaller aruco paper.

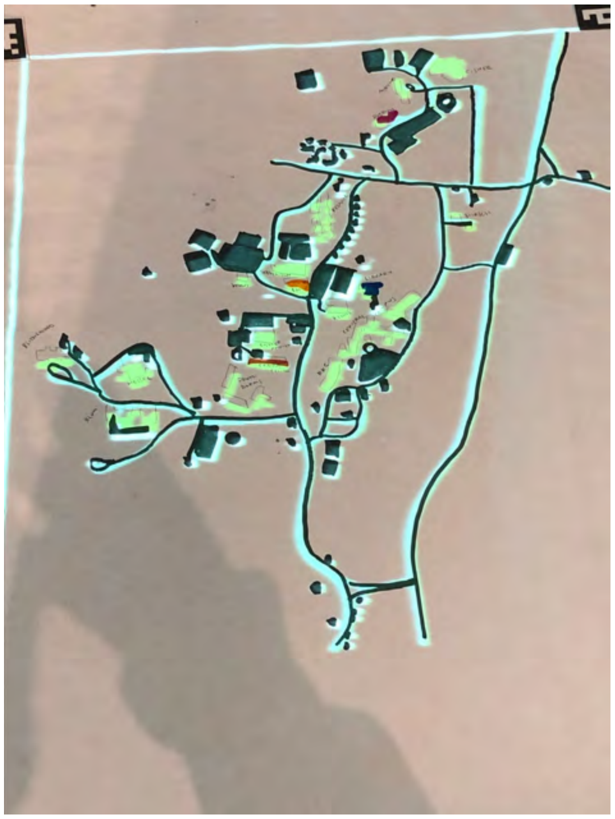
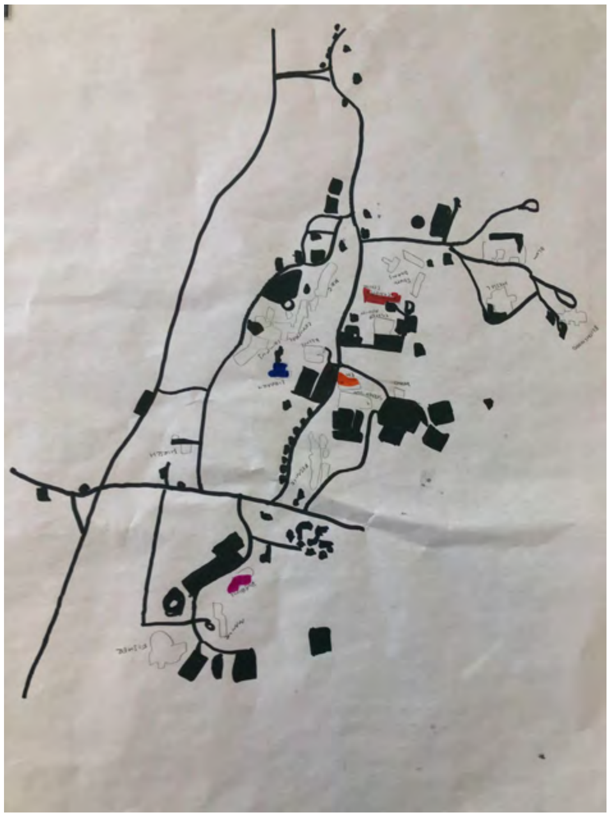




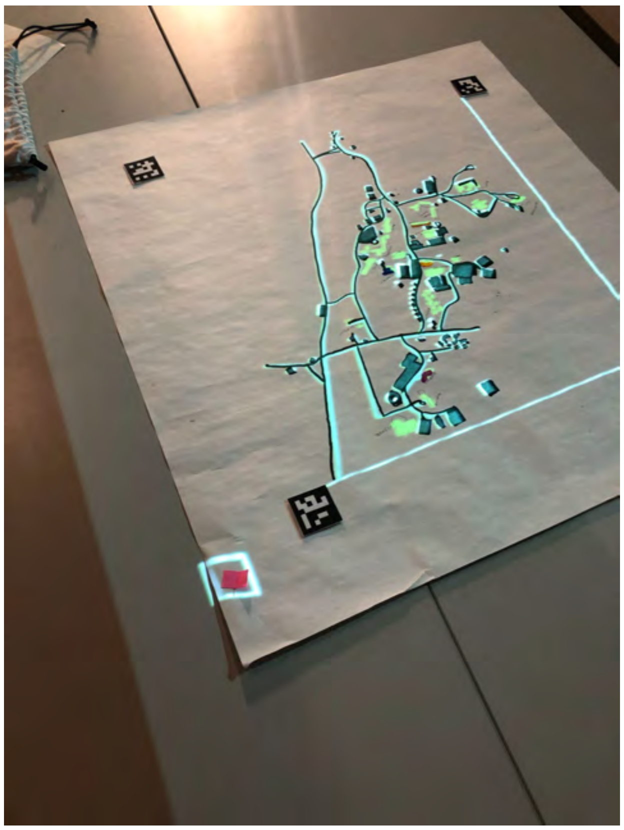
Step 12. When a location has been identified, label and trace that location on the map.

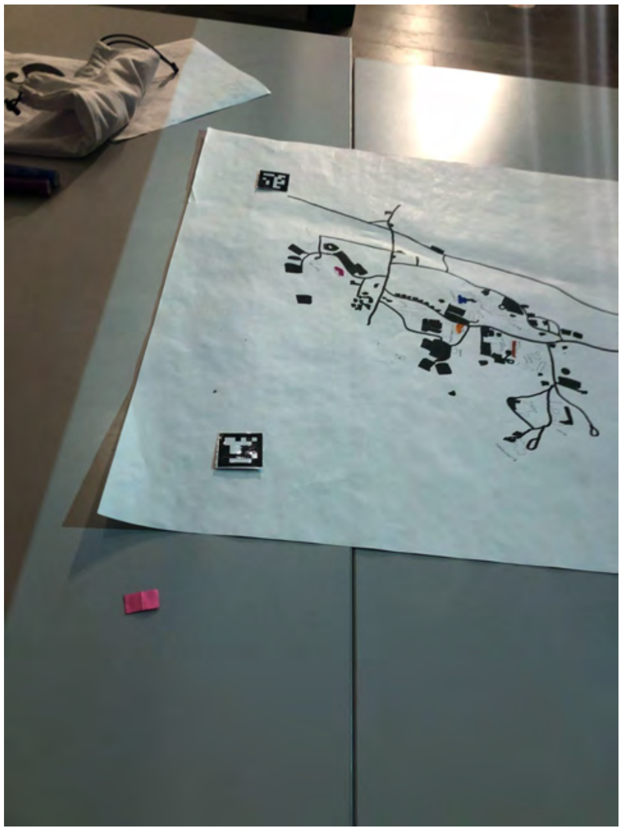
Step 13. Once all green map locations have been explored, labeled, and traced, remove the smaller ArUco paper from the frame.

Step 14. Color in the map using red, orange, blue, and purple sharpies. Red corresponds to the best place to hang out on campus. Orange represents the prettiest view, blue is the best place to study, and purple is the best dorm.

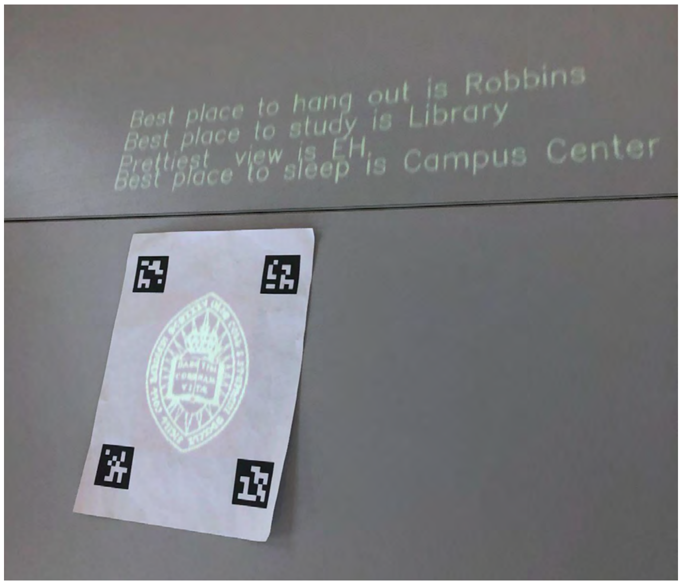



Step 15. After the map tracing and coloring has been finalized, place the pink post-it in the top left corner of the projection where a small rectangle box is displayed. This will change the projection to black and will take a screenshot of the map created.





Step 16. Remove the finished map from the frame, and replace it with the smaller aruco paper. This should display, in words, the best place to hang out, the prettiest view, the best place to study, and the best dorm based on the colors chosen from the finalized map.


Best place to hang out is Robbins
Best place to study is Library
Prettiest view is EH.
Best place to sleep is Campus Center

Appendix B: Code

aruco-tracking-perfect.py

```python
import cv2
import numpy as np
import calibr
import tracker
import intersect
import traceMode
import datetime


# red=favorite places on campus to hang out (problem, hand fits this color range)
# blue=favorite places on campus to study
# purple=best dorm
# orange=prettiest view

# used for saving screenshots for archive based on date and time
count = str(datetime.datetime.now())

# pictures and vid capture
cap = cv2.VideoCapture(1)
map = cv2.imread('data/diy-map-neon-new.jpg')
aruco = cv2.imread('data/aruco-paper.png')
calib = cv2.imread('data/pattern.png')
logo = cv2.imread('data/logo.png')

# sizing for the calibration pattern
scale_percent = 60  # percent of original size
width = int(calib.shape[1] * scale_percent / 100)
height = int(calib.shape[0] * scale_percent / 100)
dim = (width, height)
calib = cv2.resize(calib, dim)

# finding middle corners in chessboard pattern (calibration image, not live footage)
```

```
found, calibPoints = cv2.findChessboardCorners(calib, (9, 6))

# makes map the same size as the aruco paper
hT, wT, cT = aruco.shape
map = cv2.resize(map, (wT, hT))



# top left and bottom right rectangle bounds of campus buildings
robbinsTL = [490, 163]
robbinsBR = [550, 192]

fisherTL = [587, 58]
fisherBR = [634, 82]

manorTL = [522, 95]
manorBR = [549, 126]

resnickTL = [387, 332]
resnickBR = [418, 411]

stevensonTL = [331, 443]
stevensonBR = [380, 483]

blumTL = [127, 699]
blumBR = [182, 733]

hesselTL = [129, 616]
hesselBR = [181, 654]

southDormsTL = [248, 655]
southDormsBR = [330, 704]

hirschTL = [574, 372]
hirschBR = [610, 417]

kappaTL = [447, 463]
kappaBR = [474, 484]

woodTL = [274, 470]
```

```
woodBR = [303, 494]

libraryTL = [445, 503]
libraryBR = [469, 535]

klineTL = [406, 536]
klineBR = [430, 570]


sandsTL = [372, 703]
sandsBR = [387, 718]

rkcTL = [389, 634]
rkcBR = [406, 679]

fannexTL = [276, 571]
fannexBR = [290, 601]

ccTL = [294, 609]
ccBR = [346, 646]

blithewoodTL = [15, 569]
blithewoodBR = [80, 623]

centralcampTL = [443, 535]
centralcampBR = [500, 631]

ehTL = [350, 490]
ehBR = [380, 510]


# checking booleans
calibrated = False

# initializing homography variables
homogCamToComp = []
homogCompToCam = []

# initializing variables for location of building text descriptors
```

```
tlocX = 500
tlocY = 500
text = " "

red = " "
orange = " "
purple = " "
blue = " "

while True:
   # constantly captures webcam as images, setting all windows to size of callibration
image (dim)
   sucess, imgWebcam = cap.read()
   # change this to resize when the webcam is opened

   imgWebcam = cv2.resize(imgWebcam, dim)

   # moves calibration image to be on separate screen (not laptop); tweak the
'moveWindow' numbers
   cv2.namedWindow('calib')
   cv2.moveWindow('calib', -1500, 205)
   cv2.setWindowProperty(
       'calib', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN)

   cv2.imshow('calib', calib)

   # videostream on laptop
   cv2.imshow('vid', imgWebcam)

   # calibrate when 'c' is pressed, homography matricies
   if cv2.waitKey(1) & 0xFF == ord('c'):
       homogCamToComp, homogCompToCam, calibrated = calibr.calibrate(
           imgWebcam, calibPoints, calibrated)

   # after calibration, this portion sets up the projection
   if calibrated == True:
       imgBlank = np.zeros((height, width, 3), np.uint8)
       imgBlank = cv2.resize(imgBlank, dim)
       cv2.namedWindow('blank', cv2.WINDOW_NORMAL)
```

```
cv2.moveWindow('blank', -1500, 205)
cv2.setWindowProperty(
    'blank', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN)
cv2.imshow('blank', imgBlank)

# initiating blank mapProj here for color tracking
mapProj = np.zeros(
    (imgWebcam.shape[0], imgWebcam.shape[1], 3), np.uint8)

# image transformed from camera input to screen
mapTransformi = cv2.warpPerspective(imgWebcam, homogCamToComp,
                                    (imgWebcam.shape[1], imgWebcam.shape[0]))

cv2.imshow('mapTransformi', mapTransformi)

# findind aruco markers
arucoFound = tracker.findArucoMarkers(imgWebcam)
bboxList = arucoFound[0]
idsList = arucoFound[1]

# getting location of markers if they are detected
id1Index = np.where(idsList == 1)
id2Index = np.where(idsList == 2)
id3Index = np.where(idsList == 3)
id4Index = np.where(idsList == 4)

id5Index = np.where(idsList == 5)
id6Index = np.where(idsList == 6)
id7Index = np.where(idsList == 7)
id8Index = np.where(idsList == 8)

bboxListAll = []
for corners in bboxListAll:
    bboxList.append(corners[0])

# ArUco screen detection booleans
counterUno = False
counterDos = False
```

```
        # if aruco 5,6,7,8 paper is detected, warp logo to location
        if (id5Index[0].size > 0 and id6Index[0].size > 0 and id7Index[0].size > 0 and
id8Index[0].size > 0):
            bbox5 = bboxList[id5Index[0][0]]
            bbox6 = bboxList[id6Index[0][0]]
            bbox7 = bboxList[id7Index[0][0]]
            bbox8 = bboxList[id8Index[0][0]]

            mapTranDos, matrixiDos, brDos = tracker.augmentAruco(
                bbox5, bbox6, bbox7, bbox8, imgWebcam, logo)

            tlocX = int(brDos[0])
            tlocY = int(brDos[1]-100)

            counterDos = True

        # if aruco 1,2,3,4 paper is detected, warp map to it
        if(id1Index[0].size > 0 and id2Index[0].size > 0 and id3Index[0].size > 0 and
id4Index[0].size > 0):
            bbox1 = bboxList[id1Index[0][0]]
            bbox2 = bboxList[id2Index[0][0]]
            bbox3 = bboxList[id3Index[0][0]]
            bbox4 = bboxList[id4Index[0][0]]

            mapTran, matrixi, br = tracker.augmentAruco(
                bbox1, bbox2, bbox3, bbox4, imgWebcam, map)

            # mapProj is the smaller image of what is actually being projected on the
paper
            mapProj = cv2.warpPerspective(
                imgWebcam, matrixi, (imgWebcam.shape[0], imgWebcam.shape[1]))
            cv2.rectangle(mapProj, (490, 163), (519, 192), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (587, 58), (634, 82), (0, 0, 0)), 5
            cv2.rectangle(mapProj, (522, 95), (549, 126), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (387, 332), (418, 411), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (331, 443), (380, 483), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (350, 490), (380, 510), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (127, 699), (182, 733), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (129, 616), (181, 654), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (248, 640), (330, 704), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (574, 372), (610, 417), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (447, 463), (474, 484), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (274, 470), (303, 494), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (445, 503), (469, 535), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (406, 536), (430, 570), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (482, 522), (500, 570), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (372, 703), (387, 718), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (389, 634), (406, 679), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (276, 571), (290, 601), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (294, 609), (346, 646), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (15, 569), (76, 623), (0, 0, 0), 5)
            cv2.rectangle(mapProj, (443, 535), (483, 631), (0, 0, 0), 5)

            cv2.imshow('mapSmaller', mapProj)

            counterUno = True

        # if both aruco sets are detected
        if(counterUno == True and counterDos == True):

            # combine the projection warps from aruco 1-4 and aruco 5-8
            vis = cv2.bitwise_or(mapTran, mapTranDos)
            # project on the calibration plane
            mapTransform = cv2.warpPerspective(vis, homogCamToComp,
                                               (imgWebcam.shape[1],
imgWebcam.shape[0]))
            cv2.namedWindow('warp')
            cv2.moveWindow('warp', -1500, 205)
            cv2.setWindowProperty(
                'warp', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN)

            # text changes when buildings are selected
            cv2.putText(mapTransform, text, (tlocX, tlocY),
                        cv2.FONT_HERSHEY_SIMPLEX, 1.0,
                        (255, 255, 255))

            cv2.imshow('warp', mapTransform)
            # cv2.waitKey(5) ???

        # if aruco 1-4 is detected
        elif(counterUno == True and counterDos == False):
            # second warp onto calibration plane
            mapTransform = cv2.warpPerspective(mapTran, homogCamToComp,
                                               (imgWebcam.shape[1],
imgWebcam.shape[0]))
            # rectangle for screenshot purpose
            cv2.rectangle(mapTransform, (0, 0),
                          (50, 50), (255, 255, 255), 5)
            cv2.namedWindow('warp')
            cv2.moveWindow('warp', -1500, 205)
            cv2.setWindowProperty(
                'warp', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN)

            # only map projection
            cv2.imshow('warp', mapTransform)

        # if only aruco 5-8 is detected
        elif(counterUno == False and counterDos == True):
            # second warp onto calibration plane
            mapTransform = cv2.warpPerspective(mapTranDos, homogCamToComp,
                                               (imgWebcam.shape[1],
imgWebcam.shape[0]))

            cv2.namedWindow('warp')
            cv2.moveWindow('warp', -1500, 205)
            cv2.setWindowProperty(
                'warp', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN)
            cv2.putText(mapTransform, red, (tlocX-200, tlocY-80),
                        cv2.FONT_HERSHEY_SIMPLEX, 1.0,
                        (255, 255, 255))
            cv2.putText(mapTransform, blue, (tlocX-200, tlocY-50),
                        cv2.FONT_HERSHEY_SIMPLEX, 1.0,
                        (255, 255, 255))
            cv2.putText(mapTransform, orange, (tlocX-200, tlocY-20),
                        cv2.FONT_HERSHEY_SIMPLEX, 1.0,
                        (255, 255, 255))
```

```
    cv2.putText(mapTransform, purple, (tlocX-200, tlocY),
                cv2.FONT_HERSHEY_SIMPLEX, 1.0,
                (255, 255, 255))

    # only logo projection
    cv2.imshow('warp', mapTransform)

# convert minimap feed to HSV instead of BGR
hsvFrame = cv2.cvtColor(mapProj, cv2.COLOR_BGR2HSV)
# converts the overall feed, after warping to HSV instead of BGR
hsvFrameBig = cv2.cvtColor(mapTransformi, cv2.COLOR_BGR2HSV)

# bounds of the hot pink post it, determined by threshold.py
pink_lower = np.array([161, 112, 133], np.uint8)
pink_upper = np.array([179, 255, 255], np.uint8)
pink_mask = cv2.inRange(hsvFrame, pink_lower, pink_upper)

pink_mask_big = cv2.inRange(hsvFrameBig, pink_lower, pink_upper)

kernal = np.ones((5, 5), "uint8")

pink_mask = cv2.dilate(pink_mask, kernal)
pink_mask_big = cv2.dilate(pink_mask_big, kernal)

contours, hierarchy = cv2.findContours(pink_mask,
                                       cv2.RETR_TREE,
                                       cv2.CHAIN_APPROX_SIMPLE)
contoursBig, hierarchyBig = cv2.findContours(pink_mask_big,
                                             cv2.RETR_TREE,
                                             cv2.CHAIN_APPROX_SIMPLE)

# finding contours of the pink located, if only the map projection is present
if(counterUno == True and counterDos == False):
    for pic, contour in enumerate(contoursBig):
        area = cv2.contourArea(contour)
        # if enough pink is located
        if(area > 350):
            # contour dimensions
            x, y, w, h = cv2.boundingRect(contour)
```

```
                    cv2.rectangle(mapTransformi, (x, y),
                                  (x + w, y + h),
                                  (0, 0, 0), 2)
                    # top left and bottom right corners of contour dimensions
                    contTL = [x, y]
                    contBR = [x+w, y+h]

                    # if the pink location overlaps with the box in the top left
corner, take a screenshot
                    if(intersect.intersect((1, 1), (201, 101), contTL, contBR)):
                        cv2.namedWindow('warp')
                        cv2.moveWindow('warp', -1500, 205)
                        cv2.setWindowProperty(
                            'warp', cv2.WND_PROP_FULLSCREEN, cv2.WINDOW_FULLSCREEN)
                        # changes background to black
                        cv2.imshow('warp', imgBlank)
                        cv2.waitKey(5)
                        # screenshot saved to Archive folder
                        cv2.imwrite("data/Archive/mapFrom" +
                                    str(count)+".jpg", mapProj)
                        print("screenshot")

                        # screenshot read to tracemode program
                        colors = cv2.imread(
                            "data/Archive/mapFrom" +
                            str(count)+".jpg")
                        #execute traceMode
                        redOut, orangeOut, blueOut, purpleOut = traceMode.traceMode(
                            colors, width, height)

                        #change text to be projected on aruco 5,6,7,8
                        red = "Best place to hang out is " + redOut
                        orange = "Prettiest  view is "+orangeOut
                        blue = "Best place to study is "+blueOut
                        purple = "Best place to sleep is "+purpleOut


            # if both aruco papers are present, if pink intersects with tl and br building
locations, change logo to picture of location and change
```

```
        # text to be the name of that building
        elif(counterUno == True and counterDos == True):
            print("both")
            for pic, contour in enumerate(contours):
                area = cv2.contourArea(contour)
                # if enough pink is located
                if(area > 350):
                    print("enough")
                    # bounds of the located pink
                    x, y, w, h = cv2.boundingRect(contour)

                    cv2.rectangle(mapProj, (x, y),
                                  (x + w, y + h),
                                  (0, 0, 0), 2)

                    cv2.imshow('comp', mapProj)
                    # conts = np.array(
                    #     [[x, y], [x+w, y], [x+w, y+h], [x, y+h]], np.int32)
                    contTL = [x, y]
                    contBR = [x+w, y+h]
                    if(intersect.intersect(robbinsTL, robbinsBR, contTL, contBR)):
                        logo = cv2.imread('data/robbins.jpeg')
                        text = "ROBBINS"

                    elif(intersect.intersect(manorTL, manorBR, contTL, contBR)):
                        logo = cv2.imread('data/manor.jpeg')
                        text = "MANOR"

                    elif(intersect.intersect(fisherTL, fisherBR, contTL, contBR)):
                        logo = cv2.imread('data/fisher.jpeg')
                        text = "FISHER"

                    elif(intersect.intersect(resnickTL, resnickBR, contTL, contBR)):
                        logo = cv2.imread('data/resnick.jpeg')
                        text = "RESNICK"

                    elif(intersect.intersect(stevensonTL, stevensonBR, contTL,
contBR)):
                        logo = cv2.imread(
                            'data/stevenson.jpeg')
                        text = "STEVENSON"

                    elif(intersect.intersect(blumTL, blumBR, contTL, contBR)):
                        logo = cv2.imread('data/blum.jpeg')
                        text = "BLUM"

                    elif(intersect.intersect(hesselTL, hesselBR, contTL, contBR)):
                        logo = cv2.imread('data/hessel.jpeg')
                        text = "HESSEL"

                    elif(intersect.intersect(southDormsTL, southDormsBR, contTL,
 contBR)):
                        logo = cv2.imread(
                            'data/southdorms.jpeg')
                        text = "SOUTH DORMS"

                    elif(intersect.intersect(hirschTL, hirschBR, contTL, contBR)):
                        logo = cv2.imread('data/hirsch.jpeg')
                        text = "HIRSCH"

                    elif(intersect.intersect(kappaTL, kappaBR, contTL, contBR)):
                        logo = cv2.imread('data/kappa.png')
                        text = "KAPPA"

                    elif(intersect.intersect(woodTL, woodBR, contTL, contBR)):
                        logo = cv2.imread('data/wood.jpeg')
                        text = "WOOD"

                    elif(intersect.intersect(libraryTL, libraryBR, contTL, contBR)):
                        logo = cv2.imread('data/library.jpeg')
                        text = "LIBRARY"

                    elif(intersect.intersect(klineTL, klineBR, contTL, contBR)):
                        logo = cv2.imread('data/kline.jpeg')
                        text = "KLINE"

                    elif(intersect.intersect(sandsTL, sandsBR, contTL, contBR)):
                        logo = cv2.imread('data/sands.jpeg')
```

```
                        text = "SANDS"

                    elif(intersect.intersect(rkcTL, rkcBR, contTL, contBR)):
                        logo = cv2.imread('data/rkc.jpeg')
                        text = "RKC"

                    elif(intersect.intersect(fannexTL, fannexBR, contTL, contBR)):
                        logo = cv2.imread('data/fannex.jpeg')
                        text = "FISHER ANNEX"

                    elif(intersect.intersect(ccTL, ccBR, contTL, contBR)):
                        logo = cv2.imread('data/cc.jpeg')
                        text = "CAMPUS CENTER"

                    elif(intersect.intersect(blithewoodTL, blithewoodBR, contTL,
contBR)):
                        logo = cv2.imread(
                            'data/blithewood.jpeg')
                        text = "BLITHEWOOD"

                    elif(intersect.intersect(centralcampTL, centralcampBR, contTL,
contBR)):
                        logo = cv2.imread(
                            'data/centralcamp.jpeg')
                        text = "CENTRAL CAMPUS"

                    elif(intersect.intersect(ehTL, ehBR, contTL, contBR)):
                        logo = cv2.imread(
                            'data/experimentalhumanities.jpeg')
                        text = "EH"

                    else:
                        logo = cv2.imread('data/logo.png')

    cv2.waitKey(1)
```

calibr.py

```
import cv2
import numpy as np



def calibrate(vidPic, patternPoints, calibrated):
   # finds corners in the projected calibration image
   foundCorn, cornerPoints = cv2.findChessboardCorners(vidPic, (9, 6))
   # checks if enough points are found to calibrate and that both matrices are the
same size
   if cornerPoints is None:
       print("try again")

   elif(cornerPoints.shape[0] > 4 and patternPoints.shape[0] ==
cornerPoints.shape[0]):
       # two homography matrices; first is the matrix going from webcam to computer
screen, second is revers
       homogCamToComp, maskCamComp = cv2.findHomography(
           cornerPoints, patternPoints, cv2.RANSAC, 5.0)  # going from the webcam to
the laptop image
       homogCompToCam, maskCompCam = cv2.findHomography(
           patternPoints, cornerPoints, cv2.RANSAC, 5.0)  # going from the laptop
image to the webcam
       calibrated = True
       print(calibrated)
       return (homogCamToComp, homogCompToCam, calibrated)
   else:
       print("im having issues")
       calibrate(vidPic, patternPoints, calibrated)
```

intersect.py

```
def intersect(locationTL, locationBR, contourTL, contourBR):
   if(locationBR[1] < contourTL[1] or locationBR[0] < contourTL[0] or locationTL[1] >
contourBR[1] or locationTL[0] > contourBR[0]):
       return False
   else:
       return True
```

threshold.py

```
import cv2
import sys
import numpy as np

cap = cv2.VideoCapture(0)


def nothing(x):
    pass


# Create a window
cv2.namedWindow('image')

# create trackbars for color change
# Hue is from 0-179 for Opencv
cv2.createTrackbar('HMin', 'image', 0, 179, nothing)
cv2.createTrackbar('SMin', 'image', 0, 255, nothing)
cv2.createTrackbar('VMin', 'image', 0, 255, nothing)
cv2.createTrackbar('HMax', 'image', 0, 179, nothing)
cv2.createTrackbar('SMax', 'image', 0, 255, nothing)
cv2.createTrackbar('VMax', 'image', 0, 255, nothing)

# Set default value for MAX HSV trackbars.
cv2.setTrackbarPos('HMax', 'image', 179)
cv2.setTrackbarPos('SMax', 'image', 255)
cv2.setTrackbarPos('VMax', 'image', 255)

# Initialize to check if HSV min/max value changes
hMin = sMin = vMin = hMax = sMax = vMax = 0
phMin = psMin = pvMin = phMax = psMax = pvMax = 0
```

```
while(1):
   success, img = cap.read()
   img = cv2.resize(img, (700, 700))

   output = img

   # get current positions of all trackbars
   hMin = cv2.getTrackbarPos('HMin', 'image')
   sMin = cv2.getTrackbarPos('SMin', 'image')
   vMin = cv2.getTrackbarPos('VMin', 'image')

   hMax = cv2.getTrackbarPos('HMax', 'image')
   sMax = cv2.getTrackbarPos('SMax', 'image')
   vMax = cv2.getTrackbarPos('VMax', 'image')

   # Set minimum and max HSV values to display
   lower = np.array([hMin, sMin, vMin])
   upper = np.array([hMax, sMax, vMax])

   # Create HSV Image and threshold into a range.
   hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
   mask = cv2.inRange(hsv, lower, upper)
   output = cv2.bitwise_and(img, img, mask=mask)

   # Print if there is a change in HSV value
   if((phMin != hMin) | (psMin != sMin) | (pvMin != vMin) | (phMax != hMax) | (psMax
!= sMax) | (pvMax != vMax)):
       print("(hMin = %d , sMin = %d, vMin = %d), (hMax = %d , sMax = %d, vMax = %d)"
% (
           hMin, sMin, vMin, hMax, sMax, vMax))
       phMin = hMin
       psMin = sMin
       pvMin = vMin
       phMax = hMax
       psMax = sMax
       pvMax = vMax

   # Display output image
```

```
    cv2.imshow('image', output)

    # Wait longer to prevent freeze for videos.
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break

cv2.destroyAllWindows()
```

traceMode.py

```
import cv2
import numpy as np
import intersect


def traceMode(map, width, height):
    # map = cv2.imread('ar-image-mapping/data/mapArchive0.jpg')

    #coordinates of top left and bottom right points of building locations
    robbinsTL = [490, 163]
    robbinsBR = [519, 192]

    fisherTL = [587, 58]
    fisherBR = [634, 82]

    manorTL = [522, 95]
    manorBR = [549, 126]

    resnickTL = [387, 332]
    resnickBR = [418, 411]

    stevensonTL = [331, 443]
    stevensonBR = [380, 483]

    blumTL = [127, 699]
    blumBR = [182, 733]

    hesselTL = [129, 616]
    hesselBR = [181, 654]
```

```
southDormsTL = [248, 640]
southDormsBR = [330, 704]

hirschTL = [574, 372]
hirschBR = [610, 417]

kappaTL = [447, 463]
kappaBR = [474, 484]

woodTL = [274, 470]
woodBR = [303, 494]

libraryTL = [445, 503]
libraryBR = [469, 535]

klineTL = [406, 536]
klineBR = [430, 570]

sandsTL = [372, 703]
sandsBR = [387, 718]

rkcTL = [389, 634]
rkcBR = [406, 679]

fannexTL = [276, 571]
fannexBR = [290, 601]

ccTL = [294, 609]
ccBR = [346, 646]

blithewoodTL = [15, 569]
blithewoodBR = [76, 623]

centralcampTL = [443, 535]
centralcampBR = [500, 631]

#setting bounds for red, blue, orange, and purple markers
hsvFrame = cv2.cvtColor(map, cv2.COLOR_BGR2HSV)
```

```
red_lower = np.array([2, 45, 89], np.uint8)
red_upper = np.array([6, 145, 168], np.uint8)
red_mask = cv2.inRange(hsvFrame, red_lower, red_upper)

blue_lower = np.array([107, 90, 0], np.uint8)
blue_upper = np.array([162, 255, 255], np.uint8)
blue_mask = cv2.inRange(hsvFrame, blue_lower, blue_upper)

orange_lower = np.array([3, 185, 102], np.uint8)
orange_upper = np.array([14, 255, 255], np.uint8)
orange_mask = cv2.inRange(hsvFrame, orange_lower, orange_upper)

purple_lower = np.array([166, 45, 0], np.uint8)
purple_upper = np.array([179, 255, 255], np.uint8)
purple_mask = cv2.inRange(hsvFrame, purple_lower, purple_upper)

kernal = np.ones((5, 5), "uint8")

red_mask = cv2.dilate(red_mask, kernal)
orange_mask = cv2.dilate(orange_mask, kernal)
blue_mask = cv2.dilate(blue_mask, kernal)
purple_mask = cv2.dilate(purple_mask, kernal)

contoursred, hierarchyred = cv2.findContours(red_mask,
                                             cv2.RETR_TREE,
                                             cv2.CHAIN_APPROX_SIMPLE)

hangOutRed = " "
prettiestViewOrange = " "
placeToStudyBlue = " "
placeToSleepPurple = " "
for pic, contour in enumerate(contoursred):
    area = cv2.contourArea(contour)
    # if enough red is located
    if(area > 350):
        # bounds of the located red
        x, y, w, h = cv2.boundingRect(contour)
```

```
cv2.rectangle(map, (x, y),
              (x + w, y + h),
              (0, 0, 0), 2)
contTL = [x, y]
contBR = [x+w, y+h]
#if Fisher is red, then change hangout to represent, etc
if(intersect.intersect(fisherTL, fisherBR, contTL, contBR)):
    hangOutRed = "Fisher"

elif(intersect.intersect(manorTL, manorBR, contTL, contBR)):
    hangOutRed = "Manor"

elif(intersect.intersect(robbinsTL, robbinsBR, contTL, contBR)):
    hangOutRed = "Robbins"

elif(intersect.intersect(hirschTL, hirschBR, contTL, contBR)):
    hangOutRed = "Hirsch"

elif(intersect.intersect(resnickTL, resnickBR, contTL, contBR)):
    hangOutRed = "Resnick"

elif(intersect.intersect(stevensonTL, stevensonBR, contTL, contBR)):
    hangOutRed = "Stevenson"

elif(intersect.intersect(woodTL, woodBR, contTL, contBR)):
    hangOutRed = "Wood"

elif(intersect.intersect(kappaTL, kappaBR, contTL, contBR)):
    hangOutRed = "Kappa"

elif(intersect.intersect(libraryTL, libraryBR, contTL, contBR)):
    hangOutRed = "Library"

elif(intersect.intersect(klineTL, klineBR, contTL, contBR)):
    hangOutRed = "Kline"

elif(intersect.intersect(fannexTL, fannexBR, contTL, contBR)):
    hangOutRed = "Fisher Annex"
```

```
        elif(intersect.intersect(ccTL, ccBR, contTL, contBR)):
            hangOutRed = "Campus Center"

        elif(intersect.intersect(southDormsTL, southDormsBR, contTL, contBR)):
            hangOutRed = "South Dorms"

        elif(intersect.intersect(rkcTL, rkcBR, contTL, contBR)):
            hangOutRed = "RKC"

        elif(intersect.intersect(sandsTL, sandsBR, contTL, contBR)):
            hangOutRed = "Sands"

        elif(intersect.intersect(hesselTL, hesselBR, contTL, contBR)):
            hangOutRed = "Hessel"

        elif(intersect.intersect(blumTL, blumBR, contTL, contBR)):
            hangOutRed = "Blum"

        elif(intersect.intersect(blithewoodTL, blithewoodBR, contTL, contBR)):
            hangOutRed = "Blithewood"

contoursorange, hierarchyorange = cv2.findContours(orange_mask,
                                                   cv2.RETR_TREE,
                                                   cv2.CHAIN_APPROX_SIMPLE)
for pic, contour in enumerate(contoursorange):
    area = cv2.contourArea(contour)
    # if enough orange is located
    if(area > 350):
        # bounds of the located orange
        x, y, w, h = cv2.boundingRect(contour)

        cv2.rectangle(map, (x, y),
                      (x + w, y + h),
                      (0, 0, 0), 2)
        contTL = [x, y]
        contBR = [x+w, y+h]
        if(intersect.intersect(fisherTL, fisherBR, contTL, contBR)):
            prettiestViewOrange = "Fisher"
```

```
elif(intersect.intersect(manorTL, manorBR, contTL, contBR)):
    prettiestViewOrange = "Manor"

elif(intersect.intersect(robbinsTL, robbinsBR, contTL, contBR)):
    prettiestViewOrange = "Robbins"

elif(intersect.intersect(hirschTL, hirschBR, contTL, contBR)):
    prettiestViewOrange = "Hirsch"

elif(intersect.intersect(resnickTL, resnickBR, contTL, contBR)):
    prettiestViewOrange ="Resnick"

elif(intersect.intersect(stevensonTL, stevensonBR, contTL, contBR)):
    prettiestViewOrange ="Stevenson"

elif(intersect.intersect(woodTL, woodBR, contTL, contBR)):
    prettiestViewOrange ="Wood"

elif(intersect.intersect(kappaTL, kappaBR, contTL, contBR)):
    prettiestViewOrange ="Kappa"

elif(intersect.intersect(libraryTL, libraryBR, contTL, contBR)):
    prettiestViewOrange ="Library"

elif(intersect.intersect(klineTL, klineBR, contTL, contBR)):
    prettiestViewOrange ="Kline"

elif(intersect.intersect(fannexTL, fannexBR, contTL, contBR)):
    prettiestViewOrange ="Fisher Annex"

elif(intersect.intersect(ccTL, ccBR, contTL, contBR)):
    prettiestViewOrange ="Campus Center "

elif(intersect.intersect(southDormsTL, southDormsBR, contTL, contBR)):
    prettiestViewOrange = "South Dorms"

elif(intersect.intersect(centralcampTL, centralcampBR, contTL, contBR)):
    prettiestViewOrange = "Central Campus"
```

```
        elif(intersect.intersect(rkcTL, rkcBR, contTL, contBR)):
            prettiestViewOrange = "RKC"

        elif(intersect.intersect(sandsTL, sandsBR, contTL, contBR)):
            prettiestViewOrange = "Sands"

        elif(intersect.intersect(hesselTL, hesselBR, contTL, contBR)):
            prettiestViewOrange = "Hessel"

        elif(intersect.intersect(blumTL, blumBR, contTL, contBR)):
            prettiestViewOrange = "Blum"

        elif(intersect.intersect(blithewoodTL, blithewoodBR, contTL, contBR)):
            prettiestViewOrange = "Blithewood"

contoursblue, hierarchyblue = cv2.findContours(blue_mask,
                                               cv2.RETR_TREE,
                                               cv2.CHAIN_APPROX_SIMPLE)
for pic, contour in enumerate(contoursblue):
    area = cv2.contourArea(contour)
    # if enough blue is located
    if(area > 350):
        # bounds of the located blue
        x, y, w, h = cv2.boundingRect(contour)

        cv2.rectangle(map, (x, y),
                      (x + w, y + h),
                      (0, 0, 0), 2)
        contTL = [x, y]
        contBR = [x+w, y+h]
        if(intersect.intersect(fisherTL, fisherBR, contTL, contBR)):
            placeToStudyBlue = "Fisher"

        elif(intersect.intersect(manorTL, manorBR, contTL, contBR)):
            placeToStudyBlue = "Manor"

        elif(intersect.intersect(robbinsTL, robbinsBR, contTL, contBR)):
            placeToStudyBlue = "Robbins"
```

```
elif(intersect.intersect(hirschTL, hirschBR, contTL, contBR)):
    placeToStudyBlue = "Hirsch"

elif(intersect.intersect(resnickTL, resnickBR, contTL, contBR)):
    placeToStudyBlue = "Resnick"

elif(intersect.intersect(stevensonTL, stevensonBR, contTL, contBR)):
    placeToStudyBlue = "Stevenson"

elif(intersect.intersect(woodTL, woodBR, contTL, contBR)):
    placeToStudyBlue = "Wood"

elif(intersect.intersect(kappaTL, kappaBR, contTL, contBR)):
    placeToStudyBlue = "Kappa"

elif(intersect.intersect(libraryTL, libraryBR, contTL, contBR)):
    placeToStudyBlue = "Library"

elif(intersect.intersect(klineTL, klineBR, contTL, contBR)):
    placeToStudyBlue = "Kline"

elif(intersect.intersect(fannexTL, fannexBR, contTL, contBR)):
    placeToStudyBlue = "Fisher Annex"

elif(intersect.intersect(ccTL, ccBR, contTL, contBR)):
    placeToStudyBlue = "Campus Center"

elif(intersect.intersect(southDormsTL, southDormsBR, contTL, contBR)):
    placeToStudyBlue = "South Dorms"

elif(intersect.intersect(centralcampTL, centralcampBR, contTL, contBR)):
    placeToStudyBlue = "Central Campus"

elif(intersect.intersect(rkcTL, rkcBR, contTL, contBR)):
    placeToStudyBlue = "RKC"

elif(intersect.intersect(sandsTL, sandsBR, contTL, contBR)):
    placeToStudyBlue = "Sands"
```

```
        elif(intersect.intersect(hesselTL, hesselBR, contTL, contBR)):
            placeToStudyBlue = "Hessel"

        elif(intersect.intersect(blumTL, blumBR, contTL, contBR)):
            placeToStudyBlue = "Blum"

        elif(intersect.intersect(blithewoodTL, blithewoodBR, contTL, contBR)):
            placeToStudyBlue = "Blithewood"

contourspurple, hierarchypurple = cv2.findContours(purple_mask,
                                                   cv2.RETR_TREE,
                                                   cv2.CHAIN_APPROX_SIMPLE)
for pic, contour in enumerate(contourspurple):
    area = cv2.contourArea(contour)
    # if enough purple is located
    if(area > 350):
        # bounds of the located purple
        x, y, w, h = cv2.boundingRect(contour)

        cv2.rectangle(map, (x, y),
                      (x + w, y + h),
                      (0, 0, 0), 2)
        contTL = [x, y]
        contBR = [x+w, y+h]
        if(intersect.intersect(fisherTL, fisherBR, contTL, contBR)):
            placeToSleepPurple = "Fisher"

        elif(intersect.intersect(manorTL, manorBR, contTL, contBR)):
            placeToSleepPurple = "Manor"

        elif(intersect.intersect(robbinsTL, robbinsBR, contTL, contBR)):
            placeToSleepPurple = "Robbins"

        elif(intersect.intersect(hirschTL, hirschBR, contTL, contBR)):
            placeToSleepPurple = "Hirsch"

        elif(intersect.intersect(resnickTL, resnickBR, contTL, contBR)):
            placeToSleepPurple = "Resnick"
```

```
elif(intersect.intersect(stevensonTL, stevensonBR, contTL, contBR)):
    placeToSleepPurple = "Stevenson"

elif(intersect.intersect(woodTL, woodBR, contTL, contBR)):
    placeToSleepPurple = "Wood"

elif(intersect.intersect(kappaTL, kappaBR, contTL, contBR)):
    placeToSleepPurple = "Kappa"

elif(intersect.intersect(libraryTL, libraryBR, contTL, contBR)):
    placeToSleepPurple = "Library"

elif(intersect.intersect(klineTL, klineBR, contTL, contBR)):
    placeToSleepPurple = "Kline"

elif(intersect.intersect(fannexTL, fannexBR, contTL, contBR)):
    placeToSleepPurple = "Fisher Annex"

elif(intersect.intersect(ccTL, ccBR, contTL, contBR)):
    placeToSleepPurple = "Campus Center"

elif(intersect.intersect(southDormsTL, southDormsBR, contTL, contBR)):
    placeToSleepPurple = "South Dorms"

elif(intersect.intersect(centralcampTL, centralcampBR, contTL, contBR)):
    placeToSleepPurple = "Central Campus"

elif(intersect.intersect(rkcTL, rkcBR, contTL, contBR)):
    placeToSleepPurple = "RKC"

elif(intersect.intersect(sandsTL, sandsBR, contTL, contBR)):
    placeToSleepPurple = "Sands"

elif(intersect.intersect(hesselTL, hesselBR, contTL, contBR)):
    placeToSleepPurple = "Hessel"

elif(intersect.intersect(blumTL, blumBR, contTL, contBR)):
    placeToSleepPurple = "Blum"
```

```
            elif(intersect.intersect(blithewoodTL, blithewoodBR, contTL, contBR)):
                placeToSleepPurple = "Blithewood"

    return(hangOutRed, prettiestViewOrange, placeToStudyBlue, placeToSleepPurple)
```

tracker.py

```
import cv2
import numpy as np



#sets dictionary predefined as 6X6_250
def findArucoMarkers(img, markerSize=6, totalMarkers=250, draw=True):
   #convert webcam image to greyscale
   imgGray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
   key = getattr(cv2.aruco, f'DICT_{markerSize}X{markerSize}_{totalMarkers}')
   #creates aruco dictionary as being the 6x6_250 one
   arucoDict = cv2.aruco.Dictionary_get(key)
   arucoParam = cv2.aruco.DetectorParameters_create()
   #detects the markers within the greyscale webcam feed based on the aruco dictionary
specified
   bboxs, ids, rejected = cv2.aruco.detectMarkers(imgGray, arucoDict,
                                                  parameters=arucoParam)
   return [bboxs, ids]



#takes in 4 aruco markers, one for each corner, the webcam feed and the image to map
on
def augmentAruco(bbox1, bbox2, bbox3, bbox4, img, imgAug, drawId=True):
   #gets aruco 1 bottom right corner since it is in the top left position
   br = bbox1[0][2]
   #aruco 2 bottom left corner since it is in the top right position
   bl = bbox2[0][3]
   #aruco 3 top left corner since it is in the bottom right position
   tl = bbox3[0][1]
   #aruco 4 top right corner since it is in the bottom left position
   tr = bbox4[0][0]

   #height and width of the image to map
```

```
h, w, c = imgAug.shape

#corners from aruco markers
pts1 = np.array([br, bl, tr, tl])
#corners from the image to map
pts2 = np.float32([[0, 0], [w, 0], [w, h], [0, h]])
#homography matricies
matrix, _ = cv2.findHomography(pts2, pts1, cv2.RANSAC, 5)
matrixi, _ = cv2.findHomography(pts1, pts2, cv2.RANSAC, 5)
#warping of image to map as projection
imgOut = cv2.warpPerspective(imgAug, matrix, (img.shape[1], img.shape[0]))
return imgOut, matrixi, br
```