496 Capstone: AR.t

Developers: Andre Driedger and Anneliese Ansorger
Designers: Chance Galay and Chanelle Lafitte

Introduction

Defining the Problem:

Originally, we knew that we wanted to work with augmented reality, and had a pretty specific idea of how we were going to implement it. We collaborated with the design students to choose a purpose for the program that we had in mind, and eventually decided that we would make a phone app where AR is used to see what a painting will look like in your home, before you buy it. When shopping for art pieces, consumers would appreciate the ability to preview the art in the setting they wish to put it in. A mobile app, which is easily accessible to everyone with an android smartphone, can provide this service.

Our mobile Android app.

This project has two parts. When we started the project, we had decided to make a program that would use feature matching to recognize a specific image (eg. a poster or sticker) find it's orientation, and then display some kind of useful AR artifacts in the 3D space of our recognized image. We have implemented this in OpenCV, to show that we have an in depth understanding of how AR works.

The second part is the Android app. The design students prototyped screens for user profiles, buying art, as well as filtering and browsing functionality. This functionality has not yet been implemented, and we instead chose to focus on the AR screens. The user can browse through and preview different paintings and frames.

This structure gave us the flexibility to better understand the tools and processes that make AR work, and to implement a portion of the app that our designers prototyped. It also gave us the ability to start coding on day one, before we had chosen the purpose of our app.

Motivation:

Original artwork is often very expensive; being able to see how a painting will look in your home before you buy it is a huge advantage. There is no way to currently try art in your own space before you buy it, without the owner of the art, taking a risk by lending you their painting. An AR preview app on mobile devices can be used by art galleries to help them sell their art, or by consumers to shop for art in the comfort of their homes.

Augmented Reality is an emerging technology with many useful applications. This project will showcase our knowledge and proficiency in AR programming, while also allowing us to gain experience in collaborating with other professions, such as with designers. In the working world, it is common for developers and designers to work together to build applications.

Related Works:

Our designers did a precedent analysis into many similar apps. While apps for viewing AR furniture in your home seem to be gaining popularity, they only found one phone app that sells art, and allows you to see how it looks in AR before you buy it. However this app is not real-time. The user has to take a picture of the wall, measure the wall, and input the dimensions of the room, before they get to see the result. This is a huge inconvenience for the user, and our app will be a big improvement.

We also did not find any AR apps which allow the user to preview 3D AR objects on vertical surfaces such as walls, as our project intends to do. This is likely because many vertical surfaces - like drywall - lack features that we can detect. We use image recognition to solve this problem.

Audience, Users, and Stakeholders:

Artists, designers, fine artist enthusiasts, new home owners, condo builders/ designers (around the ages of 18 - 50) anyone in the age range that can competently use a smartphone. Our key is the art industry and people looking to preview their art in their homes, and decide whether it is a good fit for the space.

Fine Artists and Photographers
Demographic Profile:
They create art or take photos for a living.
Key Objective/Use Case:
They upload their art to the app, or their gallery offers the incentive and they work with them. They get more sales through the app on various works of art.

Gallery Curators
Demographic Profile:
This demographic curates and sells art for artists
Key Objective/Use Case:
Facilitate the purchase of the art once a user purchases it through the app, as well as creating curated collections of art to be purchased together.

Art Purchasers
Demographic Profile:
Owner of space to purchase and hang art
Key Objective/Use Case:
Find alluring art through app to consider purchasing. Using AR to test how the considered art looks in their space. Purchase art that fits in their space.

Main Functionality:

Both of our implementations will:

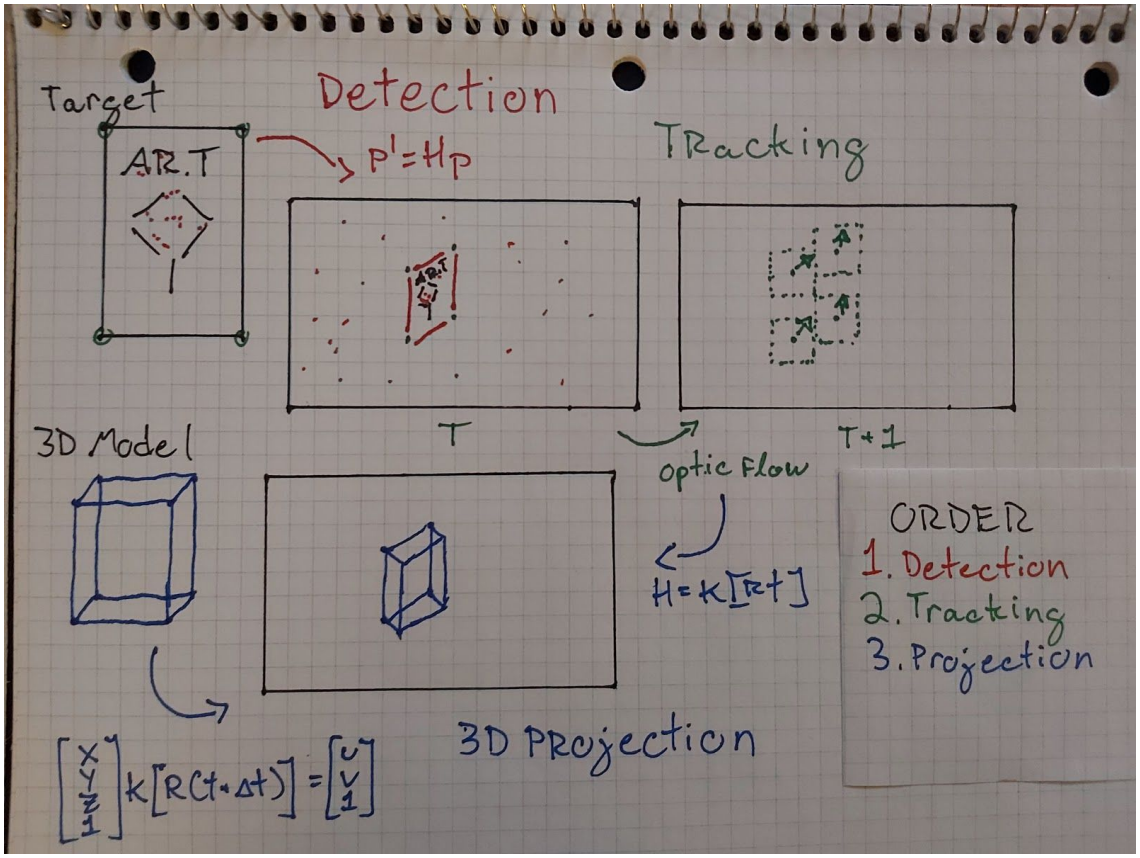
1) Be able to detect and recognize the identifying image (in our case a printable piece of paper with a unique design).
2) Be able to find the homography and rotation of the identifying image
3) Be able to track the identifier
4) Be able to project a 3D painting object (to scale) in the orientation of our identifier.

The user can also sample different frames on the painting, and move it around on the wall. As long as the camera has the target in view, it will correctly draw the AR painting in the right place and perspective.

Methods

The Big Picture:

So, how do we make AR work? First, we use Feature Detection to recognize our image and find it's homography. Once we have a good match, we use tracking to figure out where our image is from frame to frame. Then we project and draw our 3D painting into the frame.



Simple outline of our code.

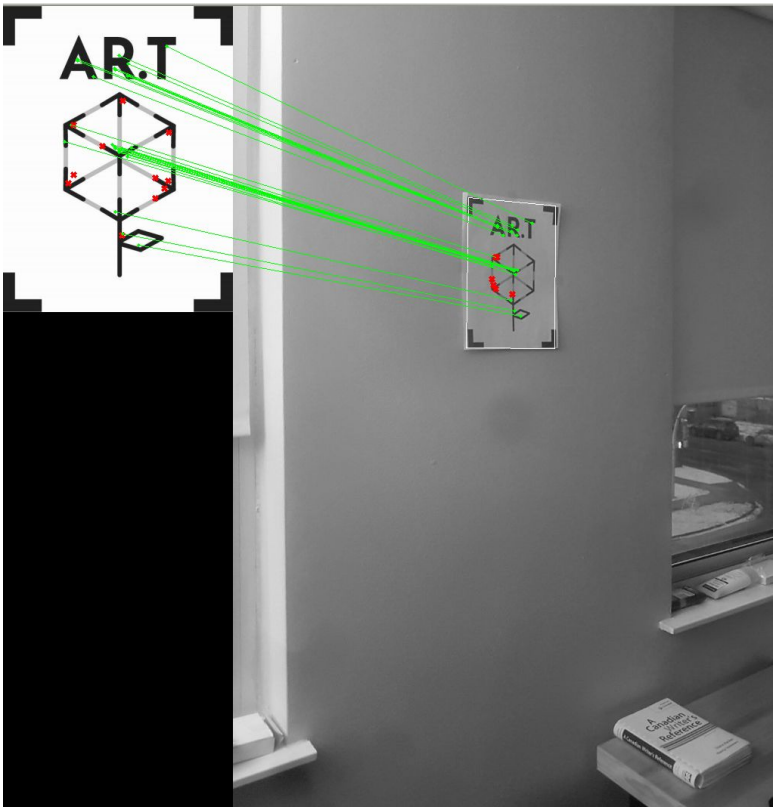
Feature Detection and Matching:

Images can be represented as waveforms in 2 dimensions (x, y) and can be decomposed into high frequency and low frequency components. Filtering images by applying a low-pass filter reduces noise, but too much filtering can remove important information. Computing the image derivative in the x and y direction can gives us good edges as well as their direction gradient. These signal processing tools are the building blocks of Feature Detectors.

So what does a good features need to have? It has to be unique; a feature detector cannot give different results when its run more than once. It needs to be scale invariant; features are detectable even when image is viewed closely or far away. Example of feature detectors are SIFT, SURF, ORB. We find that BRISK works best for our application.

The steps of our detection and matching section starts with running BRISK feature detection on both our target image and our video capture frame. Once features are detected on both images and enough of them match, we then use a RANSAC method to estimate a homography between the two images. RANSAC, or RANdom Sample Consensus will increase the impact of the inliers to generating a solution. The resulting 3x3 homography matrix (H) contains the operations needed perform a perspective transform from our target image to the video capture frame. To reduce computational complexity and increase framerate, we only perform a

perspective transform of the corner pixels of our target image.The resulting pixels on our video capture frame is then used to initialize a Lucas-Kanade Optical Flow with Pyramids tracker.



Our target image is on the left and the video frame on the right. Green lines correspond to good matches of features between both images. White lines on the frame are the outline of the perspective transform of the target image.

Tracking:

An optical flow tracker is less computationally expensive than feature detection and can be more accurate and smoother in cases where the object moves slowly. We chose an Lucas-Kanade Optical Flow with Pyramids (LK) tracker because we found it provided a good balance between simplicity and tracking performance.

To start our LK tracker we first initialize it with the four corners passed over from our detector. On every subsequent frame, the LK tracker predicts where the next tracking points are located by calculating the optical flow in a window around the initialized points. We run the LK tracker in reverse every frame to check for errors. If the pixels positions do not match each other within a margin of one pixel, we consider the current tracking phase to have failed and we revert back to detection.

Projection:

To achieve Augmented Reality on our video capture frames, we first construct a 3x4 homogeneous coordinates matrix of the 3D model we want to project. Eventually, each 3D point of our model will correspond to a 2D point in our video capture frame. We must then find our camera calibration matrix (K). We estimate this matrix by using our webcam's resolution for translations and its focal length for scaling.

Once we have K, the homography we calculated from the detection or tracking phase can be used to find the rotation and translation vectors needed for projection. We use OpenCV's built-in function cv.solvePnPRansac on our 4 tracked corner points to generate a solution for the rotation and translation vectors. Once our 2D projection is drawn, we apply a perspective transform of our cover image (the painting we want displayed) onto our 4 target points of projected 3D model. Our painting is now displayed in its correct spot within the frame.

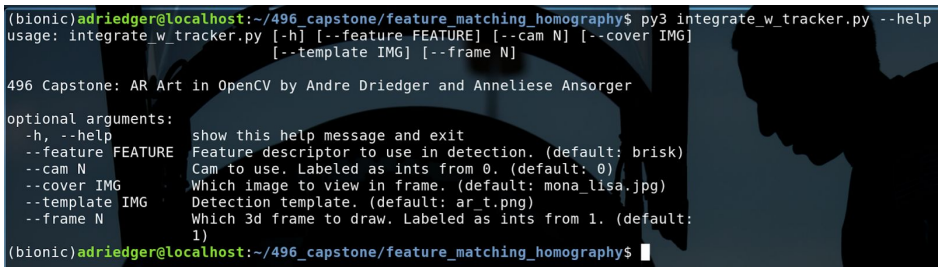
Our projection of Mona Lisa on a two-inch thick canvas-type frame.

Implementation

Our project is two parts. First, we implemented this using OpenCV in python. This is more manual, and allowed us to showcase our understanding of how AR works. Second, we made an Android app that used ARcore. With ARcore, the tools that make AR work are abstracted away, but it allowed us to use more sophisticated graphics, and to implement some of the screens that our designers prototyped.

Implementation #1:

With OpenCV, we had control over many of the tools that make AR work. We chose to work with the combination of OpenCV python bindings and Numpy because they had documentation was good and Python was our preferable language to coded. We called OpenCV functions help us perform key steps like generate key points or solve equations, and used Numpy to store our images, frames and models and perform matrix operations.



The command-line UI of our OpenCV app.

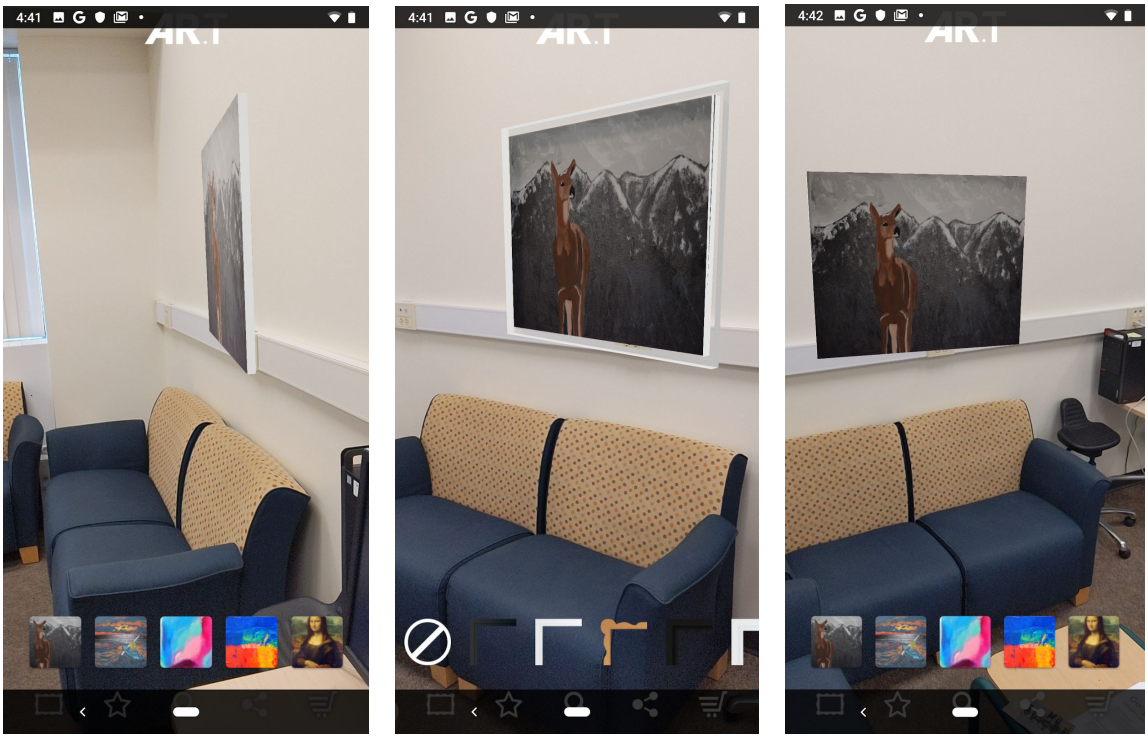
Implementation #2:

In the Android app, we had little control over the processes of feature detection, tracking, and projection. However, we did have access to more graphics functionality. We were able to manipulate many things like the texture, scale, and Ks (amount of specular lighting) using the .obj, .mtl, and .sfa files of the object.

In the code, we set up a database with the image that we want to recognize, and set up an “AugmentedImage”. ARcore uses this “AugmentedImage” to let us know when our target image has been recognized, and is being tracked. On every frame, we check to see if the “AugmentedImage” is in the TRACKING state. At this point, if we have not already put our objects in the scene, our program makes a new “AugmentedImageNode”, and assigns it to our “AugmentedImage”. The “AugmentedImageNode” stores our 3D objects, and tells ARcore where we want to place them. We can access the center pose of our recognized image, so the centre is where we set our image using an “AnchorNode”. Once we set the “AnchorNode”, ARcore will now render our 3D painting on top of our target image in every frame.

Once the above cycle is complete, we then check every frame if the “AugmentedImage” is in the STOPPED state. This means that ARcore has lost track of the image, and we remove our painting from the scene. It will now wait for the TRACKING state, and repeat the cycle listed above.

There is a little more to the story, like the code to import, load, and “build” our 3D objects in ARcore, figuring out how to use the .xml files to make and use buttons, and figuring out how to properly use gradle to include necessary files and compile the app.



Results

OpenCV:

When testing feature detection, we found that BRISK features worked best for our application. Later research showed that brisk is often used for real time applications, because brisk features are robust and almost as precise as SURF features, but are incredibly fast to compute.

We tried some methods of finding the 3D rotation based on the homography, but found that “solvePnPRansac” gave a more accurate rotation.

Overall, the app does a really good job of tracking the image and placing the object when we are up close. From halfway across the room, it makes a few more mistakes, but is still very usable.

ARcore:

We were a little disappointed that ARcore was missing some of the functionality that we expected. We thought we would be able to simply store an .obj file of a 1” x 1” canvas, and be able to scale it to the dimensions of our chosen painting. Similarly, we wanted to store the corner piece of a frame, and a one inch section of the side of the frame, and simply scale, translate, and rotate the pieces into a frame. However, there didn’t seem to be a way to access or modify the geometry of our model at runtime. We could scale our object by changing the