Jesse Weisberg - Daniel de Cordoba Gil - Tushar Mathew
CSE 559a
Furukawa
Final Project (Part 2/2)

Real-time Marker-based 3D Augmented Reality

(Part 2/2)

Abstract:

The objective of this part of the project was to integrate the fiducial marker recognition from part one and add a 3D augmented reality environment to it. To do this we combined marker recognition and facial recognition with 3D object models to simulate and render interaction using OpenGL. We used Python and OpenCV to implement these algorithms and draw on the current 2D frame, and OpenGL as a tool to render 3D objects as the final step of our augmented reality application. We were able to successfully implement this real-time system and will explain the steps we took below.

OpenGL- 3D object rendering:

To render the 3D models in a 2D image, we used OpenGL. OpenGL stands for Open Graphics Library, and it is a cross-platform open-source API for rendering 2D and 3D vectors graphics. The API allows us to interact with the GPU (Graphics Processor Unit), whose typical function is to draw the output image and display it on the screen. Its architecture is optimized to perform calculations and memory manipulation and alteration and is extremely efficient and fast and basically renders the polygons that we see on the screen in real time, allowing us to achieve hardware-accelerated rendering.

In our case, we will be using it to draw the whole scene of our augmented reality image. For this we will have to specify the position and orientation in the 3D virtual space where we want to position every 3D object, the exact textures that we will be dressing the models with and the position of lighting that we want to apply to them. From there, it automatically manages aspects of the visualization, like the superposition of a single model, where only the frontal part of the model is shown, or the superposition of various models ie. if a model covers another, the covered part is not shown. It also handles lighting, although we will not be using that feature in our project.

For our project, we split the augmented reality scene in two parts.

Part I:

The first part take care of the models we position in the scene over the fiducial markers. After retrieving the rotation and translation vectors of the markers using the glyph detection algorithm which has been explained previously, we pass them to OpenGL as the position and orientation of the models.

This is so that it can take care of their visualization. Thanks to this, when we move the glyphs around, the models follow them without hesitation and will take their exact position and orientation. In addition to this we had to slightly compensate for the the position and rotation of the glyphs, as they were referring to the top left corner of the marker and ideally we want to center the models such that the model is placed in front of the center of the marker. The following two lines of code summarize this process, thus making a correction to the translation in x and y via the function of the Z rotation about the Z axis (rvecs[2]):

```
# X coordinates
tvecs[0] += (0.5) * (math.cos(rvecs[2]) - math.sin(rvecs[2]))
# Y coordinates
tvecs[1] += (0.5) * (math.cos(rvecs[2]) + math.sin(rvecs[2]))
```

Part II:

The second part of the view is the background, which has to be mapped as a 3D object too so that OpenGL can show it in the scene. For this, we have created a giant plane far away from the camera, and we set the webcam image (modified with some rectangles for the face recognition) as the main texture of such a plane. Therefore, if we walked away from the camera far enough with our markers and our models, we would get behind the vertical plane, and the models would stop showing up in the view.

In our code, we draw each scene one frame at a time. The way OpenGL handles different frames is the following: it holds one image called the view, that is the one being shown and another one called the buffer, which is where we render every polygon, so the new polygons will not appear one after another and there will no be interruptions on the visualization. Once the image being modified in the buffer is completely rendered, it the buffer is swapped out to the display buffer so that the frame is shown on the screen. This is done with the glutSwapBuffers() function, so the rendered image is shown and the view image is stored in the buffer. This function is executed at the end of our function draw_scene(), which is the one that is run iteratively in each frame.

We made the whole application robust enough so that we can add as many models as we want. The only gating factor was the computational requirements to render the polygons in the buffer. To show this we have added several models that we thought could be useful or just fun to have: a Superman and a Batman model, a guitar, an arrow, a Darth Vader mask, and three models for hands, each for rock, paper and scissors. Each of these models was linked the recorded fiducial markers whose IDs are stored in the marker database and referenced as and when needed, so that each of them would show up when their corresponding marker was captured by the camera.

Facial Recognition:

Facial recognition is a very active area in Computer Vision and has been studied for a very long time with broad applications ranging from security, robotics, human computer interaction , digital cameras, games and entertainment. Because it has always been relevant in CV research we decided to augment our project with a facial recognition application.

Facial Recognition generally involves two stages:

1) Face Detection: In this step, a photo is searched to find any face and then image processing cleans up the facial image for easier recognition.
2) Face Recognition: here, the face that was detected in the previous step is compared to a database of known face, to decide who that person is.

For our implementation we felt that since facial recognition wasn't a big part of what we were aiming for, we decided to just implement the first stage of the process which was facial detection.

To implement this we used the help of the OpenCV library which makes it fairly uncomplicated to detect a frontal face in an image using its Haar Cascade Face Detector (also known as the Viola-Jones method). It is basically a machine learning based approach where a cascade function is trained from a lot of positive and negative images. This classifier is then used to detect objects in other images. Since we are using this for face detection, we use images of faces. Initially, the algorithm needs a lot of positive images (images of faces) and negative images (images without faces) to train the classifier. Then we extract features from it. For this, the haar features shown in below image are used. These haar features work in the same was as a convolutional kernel. Each feature is just a single value obtained by subtracting the sum of pixels under the white rectangle area from the sum of pixels under the black rectangle.

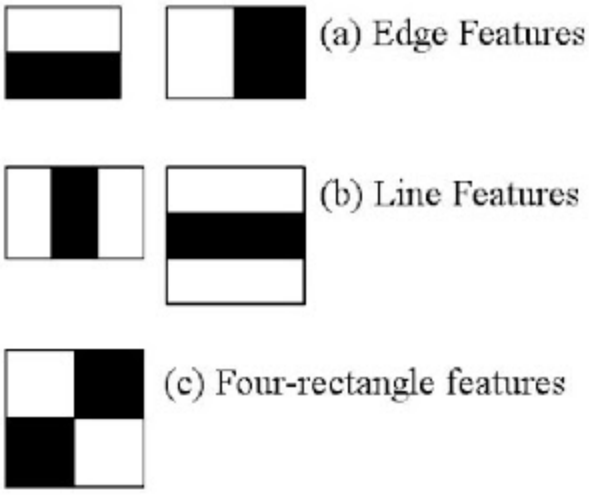

Fig 1: Haar Features

In general in any regular scene/picture image, most of the image region is a non-face region. To get around this issue of having to find it we use a simple method to check if a window is not a face region. If it is not, we discard it in and don't process it again. Instead the focus is turned to the remaining possible regions where there may or may not be a face.

For this OpenCV uses the concept of “Cascade” of Classifiers. The idea here is that instead of applying all the 6000 features on a window, the features are grouped into different stages of classifiers and then they are applied one-by-one in a stage wise manner (Normally first few stages will contain very less number of features). If a window fails the first stage, we don't need to further process it and so we can immediately discard it. We don't consider any of the remaining features on it. On the other hand , If it passes, the second stage of features is applied and the process continues. The window which passes all the stages is declared a face region.

A similar hand classifier is also used to detect hand signs based on the haar feature detection method. We felt that adding this was not much effort and we could use this to interact with the environment for future development of this project. The image below shows both these detectors operating on the frame from the live camera frame that we use.

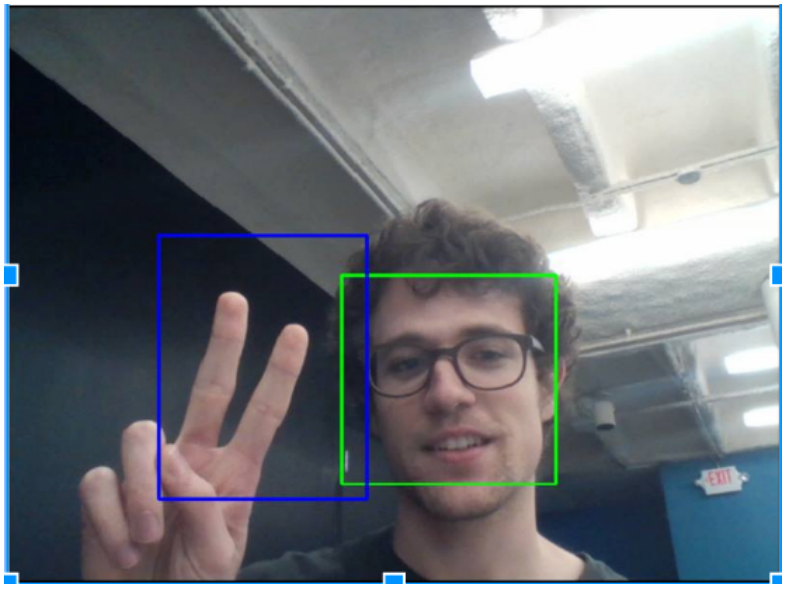

Fig 2 :Face detection and gesture detection

The last step is for us to visualize the detection and for this we display a rectangle around the face that is recognized and since we have the position of the faces(s) that we got from the multi scale detector, we can draw a rectangle around it and output it to update the current frame that we are working with. At the same time we want to be able to interact with the environment so we also initiate and pass this list as a class variable. Integration between the detected facial location and the rendered augmented reality model is discussed in the next section.

System Integration: Face Recognition, Glyph detection, and openGL

The next step is to actually build the system up from the different parts that have been described above and combine them into a single unit that functions smoothly in real time. The main challenge is to have seamless communication between the different modules and implement a sample application that interacts with a user's face or gesture and respond with an action in real time.

We had various ideas about what to do with all these features, but controlling movement in our models turned to be more complicated than what we expected, so we could only do rather simple tasks. Our first focus was showing more complex models than just geometrical shapes. We included some 3D models that were appealing and that would appear to sit on the top of our markers when we held them up in front of the webcam. To make it interesting we decided to go with Batman and Superman. We also experimented with other models, but fixed on these two models for our final version. When these models are shown for the first time, a sound is heard that relates to the specific model.

One of our primary test ideas was to use a simple model before we implemented the complex models with more triangles and aesthetic. So for this we decided to go with a simple arrow which has 6 faces and since it was an arrow, we could actually point it in a direction that we want. This led us to use the position of the face to make the arrow point at the largest face(in essence the closest to the camera) while at the same time staying right above the marker in the scene when a face was detected and if not remain at the default position. The default position is basically the position of the marker which includes both marker rotation and translation.

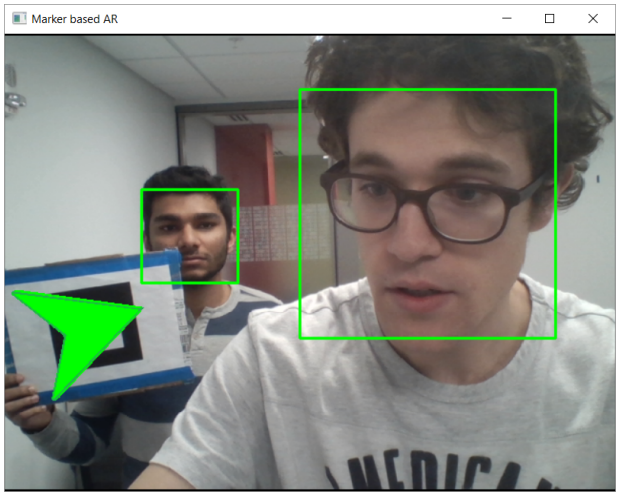

Fig 3: Person A is nearer to the camera compared to Person B so arrow points to him

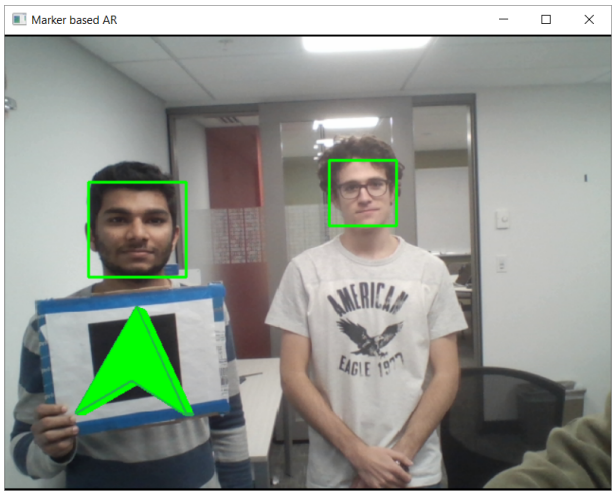

Fig 4: Person B is nearer to the camera compared to Person A so arrow points to him

Marker position:

This refers to the rotation position of the maker which we divided into four directions and named them 1,2,3 and 4. Each of these directions corresponded to the square edge of the outer polygon of the marker with a 45 degree angle on each side, thus making it cover a 90 degree region. As it is rotated the upward facing direction is considered as the marker position.

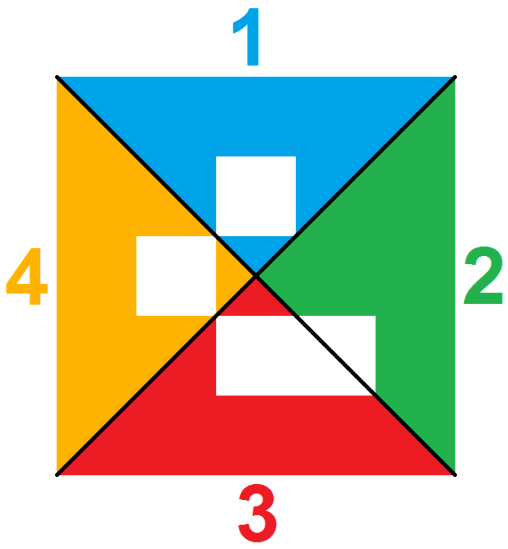

Fig 5: A sample marker showing the four marker positions

One of the primary challenges that we faced with this setup was that four 3D object models needed to be used for each of the directions and whenever the marker position changed, the new directional 3D model was put in place. However this led the model to flicker a bit at the 45 degree boundary region. To avoid this we decided we would change the translation vector based on the rotation angle and manipulate a single 3D object model in order to seamlessly rotate it in all these four directional spaces.

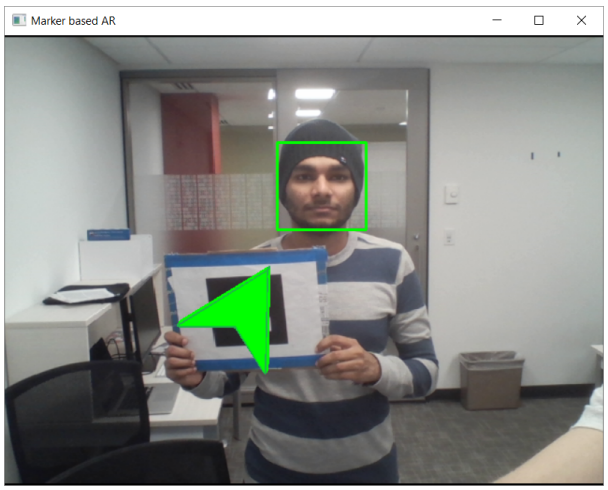

Fig 6: Person in position A, marker on left, arrow pointing to face

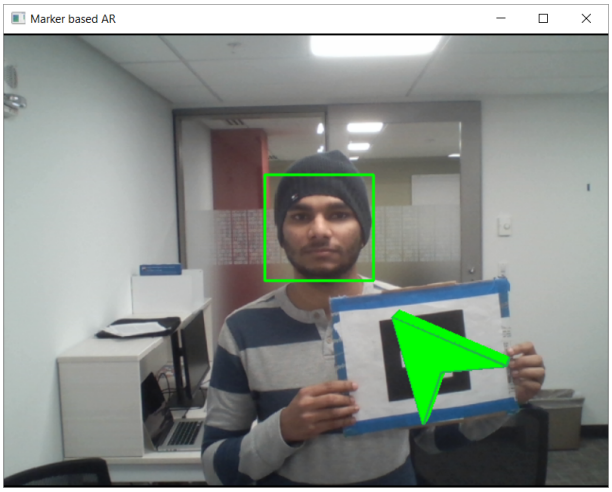

Fig 7: Person in position A, marker on right, arrow pointing to face

Sample game implementation: Rock, Paper, Scissors

To finish our implementation and make something fun and playable, we decided to implement a game that we all know: Rock, Paper, Scissors. The idea of the game with markers is pretty simple: every player has three markers, and they decide which one to play. We selected three unique markers and assigned each marker to represent a rock, paper or scissors in hand gestures. If we were to present each of these markers separately, they behave just like the other models ie. they show a clenched fist overlayed on top of the respective marker as a representation for a rock, an open palm for paper, and a 'V' sign for scissors.

Individually the system recognizes that no game is being played and so if shown individually, all of them are coded to be painted blue. This can be seen in the following figure:

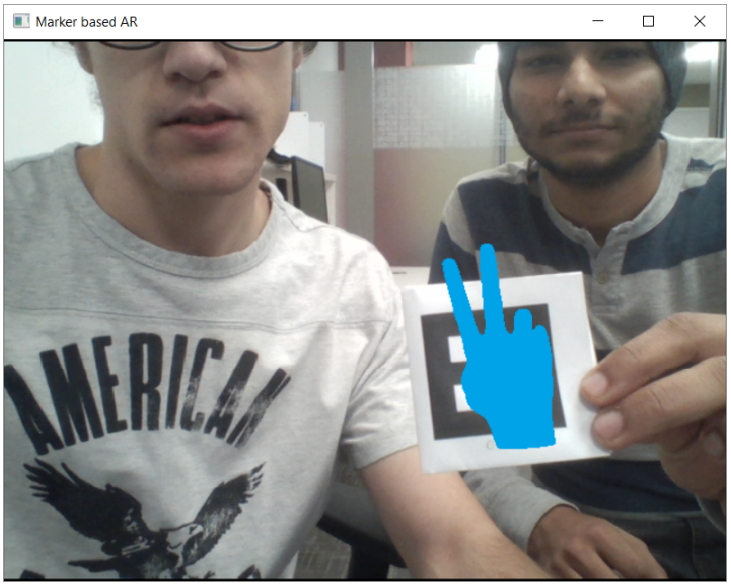

Fig 8: individual marker for game

However when we introduce more than one marker representing either of the three; rock, paper or scissors, the program understands that we intend to play the game. It then compares the three markers for a winning sign (rock>scissors , scissors>paper and paper>rock) and the winning object will be colored in green and the losing model will be colored in red. This is depicted for the three possible combinations, the screenshots of which are shown below.

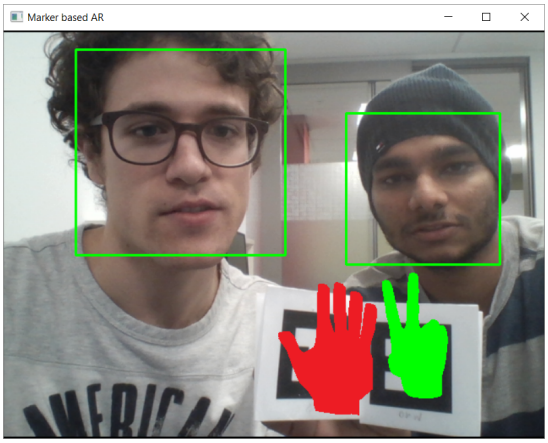

Fig 9: scissor wins

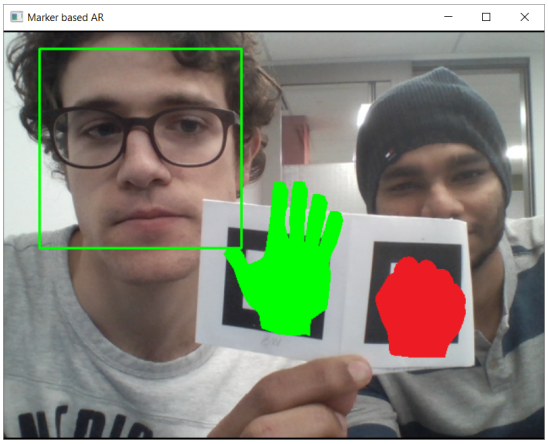

Fig 10: paper wins

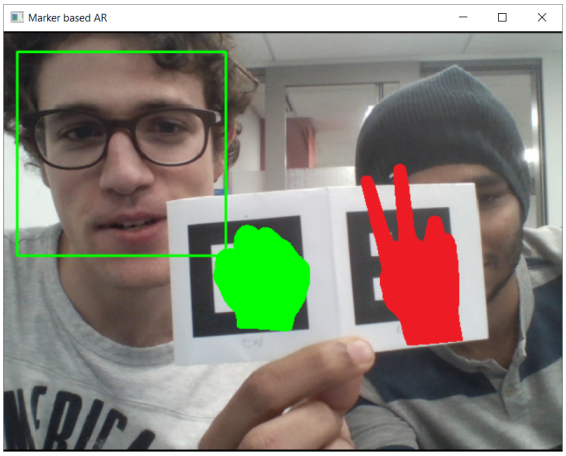

Fig 11: Rock wins

The last thing incorporated as part of the game is that if all three markers are presented at the same time, nobody wins and it is declared a tie. This is shown as all three objects colored in red.

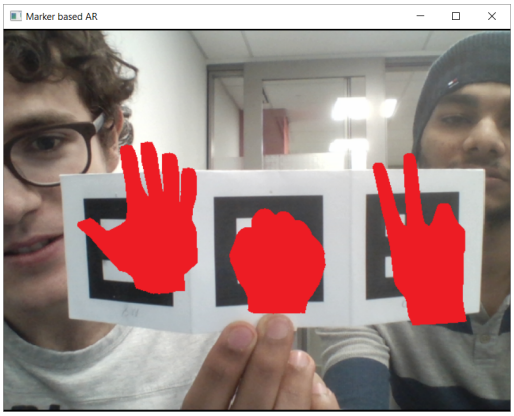

Fig 12: Tie Condition

Future work

Time permitting, we would have improved the detection of the fiducial markers. There are times when the detector loses track of the marker and our models stop showing up. This could have been improved on the detector side and we could make them more robust to the surrounding light and contrast. Also, we would have better handled these interruptions and saved the previous positions of the markers so that there were no glitches in the AR experience.

Additionally, we would have liked to improve the possibilities of our system as a gaming / exploring platform. The expressivity of our models could be improved, allowing them to move or better interact with the environment. Also, we would like to add the light rendering functionality to our platform, so that low contrast images, like the hand models for the rock, paper, scissors were more distinguishable.

Finally, we would like to add some hand recognition to be able to better interact with the models and initiate or end model behaviors with a simple gesture.

References

1. "Glyphs' Recognition." *AForge.NET :: Glyphs' Recognition*. N.p., n.d. Web. 09 Dec. 2016.
2. Heikkila, Janne, and Olli Silven. "Automatic Calibration and Error Correction." *Digital Converters for Image Sensors* (n.d.): n. pag. Web.
3. Ibe, Naoki, Atsushi Yamashita, and Hajime Asama. "Flexible Marker-based Augmented Reality Based on Estimation of Object Pose With RGB-D Sensor." (n.d.): n. pag. Web.
4. Ross Milligans blog on Augmented reality with color: https://rdmilligan.wordpress.com/2015/12/30/detect-colour-using-augmented-reality/
5. Chin-Hung Teng, Jr-Yi Chen “An Augmented Reality Environment for Learning OpenGL Programming” :2012 9th ICUIC
6. Boguet, Jean Yves. "Camera Calibration Toolbox for Matlab." *Camera Calibration Toolbox for Matlab*. N.p., n.d. Web. 09 Dec. 2016.