Reality, Augment

I have decided to do research in the field of computer vision, image processing, and augmented reality for my senior project. I have created a real time augmented reality application that has the potential for many uses. I have used many of the skills that I have acquired in my career as a computer science major.

Augmented reality is the process of adding digital data into the environment. The most commonly seen example of augmented reality is in sports, such as the current yard and first down lines you see when watching a football game. These lines look to be part of the environment but are placed there digitally. There are also many augmented reality applications that are beginning to hit the market for mobile devices. I believe that augmented reality is going to play a very large role in our lives in the near future. There is lots of research being conducted in the field, including research from Google, with their “Google glass” project. With technology such as video glasses with a built in camera, augmented reality is going to be everywhere.

Most augmented reality applications require a maker, which is something inside the environment that shows where the digital input should be displayed. One common practice is to use QR codes, which are the 2 dimensional bar codes that have started to pop up in many different places in our world. The application will detect the marker and display its digital data relative to its location. The digital data is often also affected by the angle at which the maker is seen. A popular implementation of augmented reality is displaying a 3 dimensional object “floating” above the maker. As you look at the maker from different angles you see different views of that 3d object. My project is similar to this approach but instead of the object floating about the marker, I will put in what will look like a window looking out into a scene. Much like a normal window, looking out this “fake” window at different angles will reveal different parts of the environment.

To accomplish this I am using a ray tracer program that I have created. I have integrated some of Barry Peterson’s code so that now the ray tracer has the ability to draw squares with textures on them. To create the illusion of a window I will have one invisible square that represents the window. Rays that do not hit the window squares are not drawn; instead the video feed of the environment is displayed. If a ray hits the window square, then the digital data should be displayed. Behind the invisible window I have created the scene which consists of a square with the scene texture wrap. Based on the angle at which the marker is viewed, I rotate the entire scene, so the view changes depending at what angle you are viewing the window from.

After getting the ray tracer working, the next step was to do research on image processing and computer vision. After scouring the web for tools, I discovered OpenCV, an open source computer vision library that had many useful methods that I decided would be best suited for my project. I started working with the library in python, doing image processing on still images. I spent some time working with OpenCV’s many filters.

One reason that OpenCV is so useful is that it has great documentation. The pages on Image filtering have useful and interesting information on implement the filters, as well as the mathematics and theory of how the filters are actually working. There is also lots of tutorials and example code for accomplishing certain tasks that has been very useful.

Along with filters, OpenCV has many other features that were very useful. The library gives you the ability to draw on and click images, which was very helpful because I had never done anything with graphics in either python or C++. There are also several built in data structures that were useful at times

After familiarizing myself with OpenCV I attempted to locate a marker in the environment. I had initially decided to try and detect QR codes because they have the advantage of being able to hold digital information. My first attempts used OpenCV’s built in feature detection methods. OpenCV also has great documentation and tutorials for using these methods. The first feature detection technique was to use a canny edge detector. A canny edge detector applies a Gaussian filter, and then searches the image for edges. This technique worked well with my test images under specific circumstances. If the image consisted of a QR code against a white background then it could find the edges easily. When I tested the canny detection with busy, more real world backgrounds, it would pick up too many other edges. This would make it too hard to get the necessary information out of the image. The data itself was also not in a format that was easy to manipulate and analyze, I decided that canny was not the best option and looked for other methods.

The next algorithm I decided to try was called HoughLines, which looks for lines using a probabilistic Hough transform. This approach seemed much better because it would output a vector of lines into a storage material, which I could then analyze. Houghline only works on 8-bit, single channel, binary source images so I had to convert my images to this format. In order to do the conversion I used a threshold filter. First, I created a min and max threshold scalar and gave it values for the high and low RGB values. The filter goes though the image, pixel by pixel, and creates a new binary image, a 1 if the RGB values of the image were within the threshold and a 0 otherwise. I could then give this new image to the houghlines filter and it would find lines within the image. To visualize this I drew the lines on top of the image that I was processing.

The filter did find many of the correct lines and outlined boxes of the QR code, but again it was not consistent enough. Sometime one line in the image would be represented by several disconnected houghlines, and some of the lines were missed all together. To impose its reliability I played around with some filters on the thresholded image. The first filter I used was a erode filter, which greatly reduced noise. Eroding takes a small area of pixels, known as a neighborhood, and finds the minimum values of that area, and sets the new value for the neighborhood to that value. Places where there are only small amounts of color are then removed, and with less noise the houghlines algorithm found less false lines. Unfortunately it also removed some of the values needed to find full lines of the QR code, so the lines were broken into a collection of smaller lines. I tried merging the smaller lines into one larger lines but it became too hard when there were so many lines.

I then used a dilation filter on the eroded image, which does the opposite as erode. Dilation takes the maximum value in a pixel neighborhood making lines more pronounced. It worked best after an erode, because it was not dilating the access noise, only the intended lines. This also improved the detection, and a final Gaussian filtered further increased its performance. After working with the houghlines for a while, and not getting useful results, I decided to change my approach.

I decided that it would be easier to detect circles then squares, so I changed the makers to be colored circles. Much like the houghlines, openCV also has a hough circles method, that searches the image for circles. I then do the same process of Appling a threshold, erode, dilate, and smooth filter. This method returns a vector of circles objects, that have x and y coordinates and a radius. This method worked very well in finding my circle marker.

After I could find circles in still images, I went on to see if it would work in real time with a video feed. In python it was easy to get the video feed, and I was able to track the circle with decent speed, but it was slightly laggy. The main problem that I had with this approach is that it did not work in all lighting. The way I fixed this problem was to change the threshold values to work with the specific lighting. OpenCV has built in trackbars that made testing different values much easier. By attaching the trackbars to the high and low values of the red green and blue, and watching in real time the thresholeded and filtered image, I could find the best values to work in the given lighting. I found that converting the image from RGB to an HSV image. This conversion greatly helped the detection algorithm. At this point in the project, I could track a colored circle in real time with great accuracy and speed. After trying to add more processing the program started to suffer in speed, so I decided that it would have to be ported to C++. Python, being an interpreted language, could not handle the processing power needed for this application.

There is also a specific Kinect SDK in C++ so I decided it would be the best language to use. Transferring the application to C++ turned out to be much harder than expected. I had never had any experience using the linker, so getting the libraries to work correctly was a big challenge for me. I eventually found out how to add the correct directories to the include and library directory list, and add the lib files to the additional dependencies as input to the linker. I also found that I had to download Cmake, a program needed to compile the library correctly. Finally I found that the dll’s have to be in a specific folder in order for it all to work. After doing this for several programs I found that you can create a properties page that you can make and just add it to any project.

At this point of the project I was able to track circles in real time and I also had the ray tracer working properly, but they were written in different languages. The ray tracer was written in java, I could not port the video processing over because there was no good java version of OpenCV. I could have possible rewritten the ray tracer in C++, but I decided that it would be more fun to use both programs independently and network them. To do this I would use raw sockets , I did not think that there would be performance issues because I was doing everything on local host.

Setting up the client on the java side was easy. I added a socket handler class to the ray tracer project and had it read in a loop. Creating the sever in C++ was a bit harder. I found the winsockets library, and after fighting the linker again I had completed a simple program that could communicate between java and C++. I then needed to transfer the video feed using this connection to the ray tracer. To accomplish this I would create an array of size width\*height\*3 of the RGB values of the image and send it over. On the other side I would receive the values and create a raster image. When I first started running the programs it looked fairly decent, but some of the pixel’s color where way off. In many places, a patch of pixels would have a range of different colors that were not correct. This problem took me a very long time to debug.

My first step in debugging this problem was to send a known image over the socket. I used a screenshot of the MS paint’s color wheel, because it had all values, 0-255, that I needed. The same problem occurred with this image in very specific places on the color wheel; this led me to believe that specific values were not being sent correctly. I printed out the out the values as they were received by the client to see if there were any discrepancies. I noticed that every once in a while a value greater than 255 was being sent thought the socket, which should never happen because 255 is the maximum RGB value. After comparing the sent values to the received values, I noticed that values between 128-150 were not being sent correctly, and they would be changed to different, random seeming numbers. If a 130 was sent it would randomly be transformed into a 36458 and similarly with the other numbers in that range. After doing some research I found many people with similar problems and no real good solution. The problem lays in the fact the winsocks sends a char array instead of an int or byte array. When I converted the RGB value to a char, some characters, the ones in the stated range, are special characters. They were received as special characters but had a different conversion when I casted it back to an int. the way that I fixed this problem was by setting any value received that was over 255 to a 145. This was not the most elegant solution that exists, but it worked fine without a noticeable difference in the video.

When this problem was finally debugged I had video streaming working form C++ to java in real time. The next step was to ray trace some objects on top of the video so that it looked like it was inside the environment. To make things easier, I down sampled the video image. By only sending every other pixel, I saw a great increase and speed. Also, by making the ray traced image the same size it was easier to do the translations.

I started off doing 2D augmented reality, which consists only of x, y and z translation, no rotational transformations. I get the x and y translations form the houghcircle that I find. I append their x and y locations onto the end of the array that is sent over the socket. The z axis transformation is related to the radius of the circle, after a few tries I found the correct ratio of radius and z translation. At this point I could have the raytraced image following my marker in real time, but it was very shaky. Although houghcircles found the circle easily, it was not very exact, the x, y, and radius values would vary slightly different each time. To fix this problem I decided to create my own circle detector.

I used hough detection to find the circles initially and then use those general locations on subsequent frames with my own algorithm. I would start by looking at that same location, with the notion that the circles would not move far from its previous location. If the binary value form the thresholded and filtered image did not equal 0, then then that location is contained in the circle. If the value was 0 then there was no circle. In this case, I would check the locations near that spot, within 200 pixels, for a non-zero pixel. After we find a pixel in the circle, we need to find the center of the circle. To do this I would send out a ray in the positive and negative x direction to find the middle of the circle x, at that midpoint I would send out rays in the positive and negative y direction to find the middle there. I would perform this action 2 more times to make sure the center is correctly found. I would then send out rays in multiple directions from that middle point to find the radius. This method would find the circles with much better accuracy then the houghcircle. If my circle detection cannot find a circle within the neighborhood of the past circle, it will go back to hough circles until it can find a new circle.

Now that I had all of the translations working correctly I started working on rotations. I started out implementing rotation around the z axis. Unfortunately, because I was using a circle as a marker I could get no information about the rotation. To fix this problem I added another circle to the detection. This second circle had to be a different color; otherwise it would interfere with the detection of the first image. To accomplish this I had to create a second thresholded image and a second houghcircle detection. Because of the way I had set everything up, finding this second circle was quite easy. When I found the second circle, I calculated the angle between the y = 0 line and the new circle. I send this value across the network, and now I can rotate objects around the z axis.

The last two things I needed to implement for this project were x-axis rotation and y-axis rotation. I decided that rotation in the x, y, and z rotation, at the same time, would be too hard, so I constrained the project to handle only x and y, or z rotation. In the situation where there was only y rotation, the difference between the lowest and highest points(dy) of the circle would remain the same, but the different between the x (dx) would get smaller the more it was rotated. To find the y rotation, I would take the ratio of dx/dy. This would give me a decimal between 0-1 that I would multiply y by 255 so that I could send it to the client as a char. In the case were there as only x-axis rotation I would do the opposite send over 255\*dy/dx. Appending these numbers to the sent message I could do translations in all three directions, and rotations on all of the axes individually.

From there I wanted to be able to do rotation on the x and y direction simultaneously, but the current implementation would not work, because both the dx and dy would be changing so the ratio would find the wrong rotation. A rotation in both the X and Y axis results in the markers looking like an angled oval. Instead of using the ratio of dx/dy and dy/dx, we need to use the ratio dx/maxOvalRadius and dy/maxOvalRadius. To find the largest radius of the oval, I sent out 25 rays from the center of the oval in different directions and found the largest one. The new ratios made it possible for there to be both x and y rotation. The next problem I encountered was with only 2 circles, I could not tell what direction the x-axis rotation was, weather forward or backwards. To fix this problem I added a third, different colored, circle to the marker at a right angle with the other two. With the third circle I needed to implement another Thresholded image and find new houghcircles, but with the structure of program, it was easy to implement. Now, if the third (bottom) circle was larger than the center circle, i could deduce that the marker was rotated away from the camera and translate that to a negative x translation. I applied the same process with the second (left) and center circle for direction dependent y rotation. In the end, the augmented reality application worked quite well.

Overall, I believe I was very successful in my development and research in computer vision and augmented reality. I can find and track a marker in real time, as well as draw 3D objects in the environment. I learned a lot about using different libraries and the difficulties that visual studios will pose on you. I enjoyed having the opportunity to work on a project that was of great interest to me.