CS415

Professor Wang

Team 21: Ryan Magdaleno, Aaron Willming, Semih Kesler, Jonathan Hung

12/13/24

Final Project Report: Mania–Vision

Computer Vision is the field of trying to obtain the "story" behind an image, as well as applying that information to other processes. I find Computer Vision to be a very interesting field. At first glance, it may appear to be a relatively narrow area of study. One might wonder how much there is to do in the world of image processing and analysis, but a closer look will reveal that there is a wide range of ways the concept can be applied to all sorts of related subjects. Such as facial detection, recognizing and reading text, special effects, self-driving algorithms; Computer Vision isn't just one simple thing but rather an entire school of related ideas and concepts built upon the same foundation.

Computer vision has become more prevalent and visible within our lives, and this class was an opportunity to understand how this technology works on the inside. For instance, Google has for some time now given you the option of uploading a photo in your browser to look for similar results. Before taking this class, I was very curious about this: How is Google taking my photo and accurately finding similar results? How is this possible? It wasn't until I took this course, and we started talking about feature detection, that I understood how this is possible. This was a very eye-opening course, where I had the opportunity to understand more how the things we see and deal with on a daily basis (like a simple search on Google) work behind the scenes. Just like this experience, I started looking at things that involved computer vision from a different perspective; a perspective with more information and curiosity than before.

Personally, I was very excited to take this course. I have a lot of experience with photo and video editing in my spare time, and I was excited to learn more about the processes behind the software I would use so frequently. I was also hoping to learn some new processes and techniques that I could use in the future to aid me in my photo and video editing endeavours. From the perspective of another team member, I was motivated to take this course because I was working with the arduino and various other electrical components and was interested in implementing computer vision to make a face tracking and following robot and also object detection device.

Ultimately, I think this course was a valuable experience that taught me a lot about photos and the algorithms used to detect things within them. I remember being very excited to learn the implementation behind how even the basic blur and sharpen edges filters work that I was so familiar with using in my image editing softwares already. It was very interesting to learn the computation behind many of the filters and algorithms that I already had lots of experience using. I enjoyed learning about the different kinds of detection algorithms, detecting corners and features. Surprisingly to me, I actually ended up really enjoying learning about the different classifications of transformations, and learning how to apply them mathematically with transformation matrices in Mini Project 4. It was an interesting mathematical puzzle to figure out how to get it to work, and required me to think about coordinates and trigonometric functions in a way I hadn't had much experience with up until that point.

One of the things I found very interesting is how computer vision algorithms, or at least the ones we explored, are very dependent on math and linear algebra, especially the modifications and operations of matrices. For instance, as in the case of the convolution operator and applying a kernel over an image, each pixel of the image where the filter is applied is

basically converted to a linear combination of its neighbor pixels based on the weights specified on the kernel. Also, it was interesting to see how different kernels, with different weights, produce different resulting filters, even though you are using the same sliding algorithm technique for all these different kernels. For me, it was just surprising how much you can accomplish by just changing the weights in a kernel, producing different pixels based on different weights of their neighbor pixels.

Overall, the course was very valuable and interesting to me. In general, I would really like to study everything we learned more, because I feel like we learned about a lot of complex topics that can be researched ever further. However, in particular, I hope to study more about feature detection, as well as image analysis and classification. It seems like an incredibly versatile and interesting facet of Computer Vision, and I feel like I've only scratched the surface of understanding it all. It feels like the part of Computer Vision that has the most relevance to what I'll be doing in the Computer Science field going forwards. As a result of this, it seems incredibly fitting that our project turned out to be so focused on image recognition and feature detection, given my noted interest in the subject.

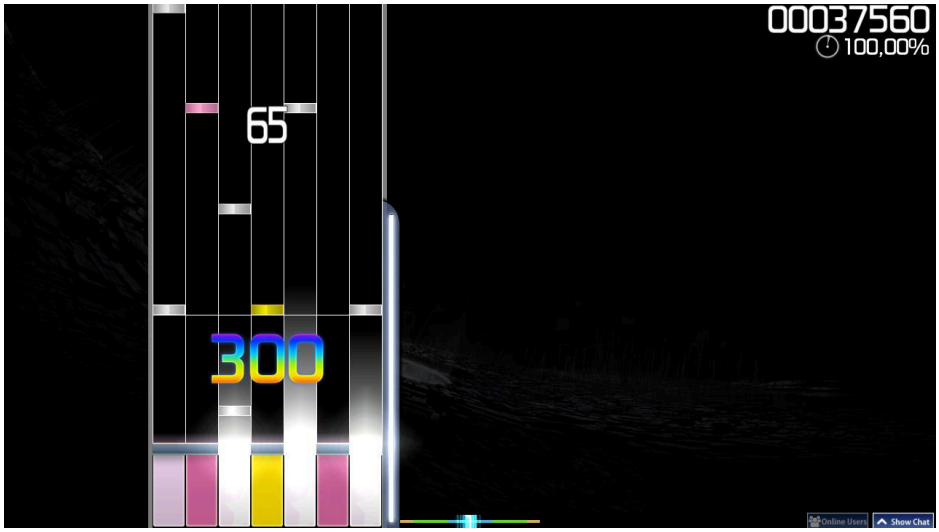
Problem and Goal

Our project, Mania-Vision, exists to be an alternative to the traditional means of recording player input, which is directly reading the input of the player with an external program simultaneously while the output program or game is running. This kind of software is often used to play back a specific sequence of computer inputs, often in some kind of video game. While the technique of recording direct input from the user with an external software works mostly well as is, there are some scenarios where it isn't quite perfect. For example, if your computer has

performance problems, and can't handle recording multiple inputs per frame in an external software while also running a complicated game where even the slightest bit of lag can mess you up, then having to run simultaneous programs on your computer to record your input just isn't a viable option. Or, if you want to analyze the gameplay of one of your favorite streamers, and only have access to footage of them playing through a keyboard or mouse cam, but not actually their inputs themselves, then that method also wouldn't work for you. Our goal with this project was to create a program that allowed users to upload a video of someone inputting in a sequence of inputs, and then the python program would be able to analyze those inputs and output a file that could then potentially be uploaded to an automatic hotkey program that would play back all of those recorded inputs in sequence. This would allow users more options and could potentially be a fitting solution to the scenarios mentioned above.

For this concept, we ultimately decided to focus specifically on rhythm games. Rhythm games are a genre of video game that challenges players to interact with the game in time with a musical beat. Players are typically required to press buttons, hit notes, or perform specific actions in sync with an on-screen prompt and the rhythm of a song. Success is often measured by timing accuracy, and gameplay can involve instruments, controllers, or touch-sensitive devices.

The reason we chose rhythm games as part of our project is that these games tend to use very few buttons, which would make them an easy target for video analysis, while rhythm game levels themselves often require the same set of precise inputs every time, and contain some internal grading system that would work well to allow us to judge the accuracy of our program. The idea as a whole isn't specifically limited to rhythm games, and can be expanded to other things, but for the vast majority of this project we will be focusing on rhythm games for their simplicity of gameplay and effective method of judging accuracy.



To be specific, we will be focusing on the rhythm game osu!mania, a game that uses a fast scrolling array of different “notes”, each with its own corresponding key. The notes need to be pressed in time with the music, with the specific presses indicated by the onscreen scrolling note interface. The speed of the notes is determined by the tempo of the song in the level. Notes often need to be pressed in quick succession, but slower songs also exist. We expect the slower songs will be significantly easier for the program to complete than the faster ones, but we will test the program on a variety of song and level types to see how it holds up under precision. osu!mania levels also have a variety of input types. It’s common for levels to use 4 keys, but some use as many as 18 different keys at once. For simplicity, we’ll focus mostly on the 4-7 key levels, but we’ll be sure to leave room for expansion to a greater number of inputs down the line.

In summary, for the goal of our project and the problem we are trying to solve, we will be working with rhythm games (in this case, osu!), and using a video of the user playing the game. The computer vision task will be to track the player’s fingers and movements in the game. After tracking the player’s fingers and evaluating which keys are pressed, the goal will be to reconstruct these keys (or the movements made by the user) and the order in which they were pressed, and store them in a separate file that can be used to reconstruct the game played by the user.

Design and Implementations

Onto the implementation of our programs, the project is split into two programs. The source code can be found in src/cli at this link: https://github.com/typeRYOON/Mania-Vision. (There is also a GUI version, but I will not go into detail about that, as it's currently a work in progress). There is a computer vision python program that generates a hit map based on an input video, and a C++ program that reads in a hit map file and reproduces the hits in real time.

The input into our python program is a video of a front view of a person playing on a keyboard. Specifically, the player's hands should be in good lighting, both hands should be decently away from the video's edges, and the camera should be slightly above the keyboard / fingers looking down. Take this video as an example: https://files.catbox.moe/6srdeb.mov. Originally, we had used a video that was more angled top down, which worked well at tracking a hand in isolation, but when paired with the keyboard it found it difficult to detect the correct landmarks for angle calculations. We found that this front-facing perspective worked the best at identifying each finger consistently and independently. Our basic vision program is tracking four fingers; the middle and index fingers on both hands. Our program makes use of Google's MediaPipe "hand landmarks" machine learning model. There are three main steps in our process chain, a finger coordinate collection step, calculation of where in the source video the four pressed points are, and finally, the hit encoding phase.

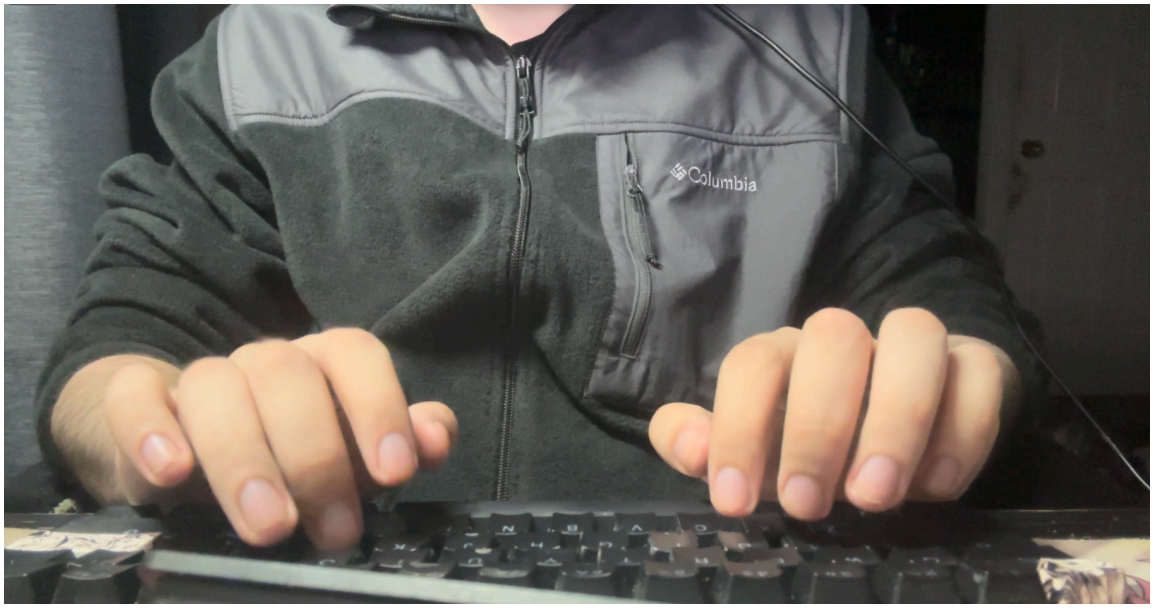


Photo: This is a frame from the input video. It shows the perspective of the camera.

There is a step 0 as well, where we read in config data from “config.ini” to be used in our program. We won’t easily be able to know which specific keys are being pressed via the video, so the first four non-space characters are for indicating what keys should be pressed, something like “S D K L”. The final non-space character(s) is for modifying the press-threshold, more on this in step 2, which talks about calculating press points for each finger. If config.ini doesn’t exist, one will be generated with “S D K L 10” as the defaults, this means starting from the left middle finger, we are pressing keys S, D, K, L, with a 10px press point threshold y shift. The key press characters are then transformed into virtual key codes; ‘S’ turns into 0x53 (83), for example. These virtual key codes are important, as they are going to be placed in the header of our vision program’s output file to be used by our presser program.

Onto step 1, the finger point coordinate collection phase. We need to collect where our fingers are in each frame of the video. The best way to do this is to check where each finger tip is on each frame as a 2D coordinate. As previously stated, to do this we’re using Google’s

MediaPipe hand landmarks model. Using OpenCV, we can create a video capture object which allows us to go through the video frame by frame and analyze each one individually. Before analyzing the frames themselves, we need to use the video capture object itself to get the FPS of the video, and save it as a float. This will become more relevant in step 3. We're also going to save the width/height dimensions in advance for step 2. Moving on to the actual frame analysis, we can use mediapipe.solutions.hand.Hand(...). It comes with two threshold fields, "min_detection_confidence" and "min_tracking_confidence". "min_detection_confidence" is how much confidence the mp_hands process function needs to state whether there are hands in the frame. A lower value may allow for slight misfires, but for our video, which we know will have hands in nearly every frame, we want as little failure to detect frames as possible. Because of this, we hard coded it to be 0.1, which is quite low, in order to ensure we don't erroneously discard frames containing hands. "min_tracking_confidence" is the confidence level needed by the model to have to be certain we have tracked a hand. Higher values increase the robustness of the model's solution, so we hardcoded it to be 0.85. We can process a frame by using OpenCV's video capture object and MediaPipe's process function together like so;
results = mp_hands.process(cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)), the parameter "frame" is a single image frame from the input video's VideoCapture read function. The "results" object has some attributes we can check in the instance that there was a success. "results.multi_hand_landmarks" is a collection of detected/tracked hands, where each hand is represented as a list of 21 hand landmarks and each landmark is composed of x, y and z components. x and y are normalized to [0.0, 1.0] by the image width and height respectively. "results.multi_handedness" is a collection of handedness of the detected/tracked hands (i.e. it tells us whether it is a left or right hand). Each hand is composed of a label and score. The label is a

string of values, either "Left" or "Right". Using these two iterable objects, if they aren't of “None” type for the current frame, we can be sure at least one hand was detected by the MediaPipe process function. “results.multi_handedness” can have at most two elements in its list, “results.multi_handedness[0]” is the first possibly detected hand. We can retrieve the hand label to determine which hand it is by retrieving the label with “results.multi_handedness[0].classification[0].label”. This will result in a string of either “Left” or “Right”, which we can convert like so: “hand = 1 if handedness == ‘Left’ else 0”. The MediaPipe model uses 21 3D hand-knuckle coordinates as the output fields. For our program, we only require the middle / index finger tip point coordinates, which are 12 (MIDDLE_FINGER_TIP) and 8 (INDEX_FINGER_TIP) respectively. To retrieve a coordinate, we can do the following; first, we need the MediaPipe solution x and y coordinate. If we loop through the “results.multi_handedness” object, our loop element, which we’ll call “landmarks”, will be an object containing the current hand’s landmark data. This loop should also include the handedness classification above. The x coordinate of the current hand is therefore obtained with “landmarks.landmark[12].x”, with “landmarks” referring to the loop object, which contains an attribute called “landmark” which allows us to index for the point we desire from the MediaPipe solution. Remember, however, that this is normalized between 0.0 and 1.0, so we will need to multiply it against our video’s width for the x coordinate and height for our y coordinate. A complete coordinate looks like this:
“(int(landmarks.landmark[12].x * width), int(landmarks.landmark[12].y * height))”. As previously stated, the index 12 refers to the tip of the middle finger on the hand. An index finger would be the exact same process, but with the index of 12 replaced with an 8. If a hand was detected, then both of these fields should exist. Within a frame, either 2 coordinates or 4

coordinates will be detected. There'll be 2 coordinates if there was only a single hand detected, and 4 coordinates if both hands are detected. If we use the handedness attribute in conjunction with this, we can collect all the tip points for each hand on every single frame. Occasionally, MediaPipe will fail to detect one of the hands in the video, usually just due to the hand being oriented at a weird angle for a brief moment. The program is expecting to see two hands, so in the instance that one of the hands isn't detected, we will simply use whatever the previous detected point was. We store the previously detected coordinate in a variable that, at the start of the program, is initialized as the coordinates (0, 0) (close to the video's edges), which will be filtered out later if used in the collection phase. The data structure used to store all of the coordinate information is a list of four lists, where each inner list represents one of the four fingers, and will contain n coordinates, where n is the number of frames found in the input video.

Onto step 2, we need to make this data useful by trying to determine which of the coordinates actually counts as a key press. We will be using our collected points to determine what the coordinate range for a key press should actually be. So, the output of this phase is four coordinates, representing the press points. First, we must filter our four lists of coordinates to remove any potential outliers. We will do this by filtering out points that are within 1/12 of the video's edges. For optimization, we will also only be using the bottom 75% of points, removing the top 25% of points from our calculation, because they are unlikely to contain any key presses. The most important information gained from these coordinates is the y level of our pressed point after all, so we can remove any that we can be confident don't contain presses. To clarify, when I refer to "bottom", I mean from our perspective, not the OpenCV coordinate system where (0, 0) is the origin and would actually be considered the top left of the video. We will now simply average out the remaining points for each finger list together, on both the x and y coordinates. Averaging

like this will result in 4 discrete “press coordinates”, one for each finger. This is where the press-threshold from our step 0 config file comes into play. We can add the press threshold to the y coordinate here, which will move our press coordinate down however many pixels were given. So, if the press-threshold was 10, and a press coordinate was (200, 800), for example, this would result in (200, 810). It lowers (from our perspective) the press coordinate y level, you can also make the press-threshold a negative value to raise the press y level. Because we are doing this averaging out independently from each finger, we will most likely get four different y levels in our press coordinates, which is good, because in an actual input video, you would expect to see the fingers not necessarily all pressing down at exactly the same y coordinate in a straight line. The user may be pressing down more forwards or backwards on some keys than others, the video might not be perfectly level, there could be subtle warping effects caused by the camera’s lens, and some fingers might not get lifted as high off the keyboard as others. Determining each press coordinate independently of the finger allows for greater flexibility. Once we’ve determined our press coordinate, we can build a “press range”, a small area of pixels in the frame that, if a coordinate is inside of, signifies a press has occurred. Our press range is bounded on the top by the press coordinate, and extends out 70px on both sides. Any coordinate that falls into its “press range” will register as a key press for that finger.

Image: a visual representation of the “press ranges” generated by the program

Onto the final step, step 3, where we must generate an output encoding file. Given a video with n frames, we will have to create a file with n press encodings. A “press encoding” is a sequence of bits which stores, for a given frame, all four fingers’ press states. So, if we have four fingers, the first four bits are what encode those four fingers’ press states respectively. The left middle finger is encoded by 0b0001, left index is 0b0010, right middle is 0b0100, and the right index is 0b1000. For instance, a press encoding of 0b1011 means that for the given frame, all fingers should be pressing, except for the right middle finger. Using this, we can generate n encodings for every single coordinate collected on each finger, using the previously determined press ranges. As previously stated, the press range is -70px to +70px around the x press coordinate. Therefore, so long as abs(example_x - press_x) < 70, the given coordinate will fall within the press range horizontally. For the vertical range, as long as it’s not less than the y press level (as “less” would indicate “above” in OpenCV’s coordinate system), it’s within the press range vertically. An example of this calculation for a finger could continue like this: Let’s say “example_coord” is at (200, 800) and the “press_coord” for its respective finger is at (210, 780). For the x range check, we perform the calculation abs(200 - 210), which indeed is within the 70px hardcoded limit. For the y range check, we check if 800 < 780 (our press_coord). It’s false in this instance, indicating that we are lower than the y press coordinate. Using the bitmap encoding from above we can generate n encodings representing each fingers press states on each of the frames.

Continuing step 3, we need to generate the text output file. Before printing all n encodings, we will first generate a header. As mentioned in step 1, earlier we stored the FPS of the video, which will be needed to tell the presser program how many lines indicate 1000ms, so it

can play the output in time. However, videos can be encoded with non-integer frame rates like 59.9958205628 for example. Over time, this will result in drift, where the output presses become desynced from the game itself, which is not good for a rhythm game. Over 10 minutes, this would result in a ~2.5 frame drift (60fps drift), which is about a 41.6ms drift. That might not sound like a lot, but in a rhythm game it can be the difference between hitting every note and not hitting any. Therefore, we will store the frame floating point value as a fraction in our encoding file's header. Specifically, it will be stored as two integers, one representing the numerator and one representing the denominator. For instance, 59.9958205628 would get stored as the integers 215325 and 3589. In the header, we will store the virtual keys from step 0 as well. For the rest of the file, we will store a press encoding for a frame on each line.

We will quickly go over the presser program. It's not part of the vision process, but rather uses the output from it. We have a parse phase where we open the encoding file, get the fps representation and virtual keys. We will create four vectors of where each element represents the start and end time of a press, in frame time. So, a pair of (233, 256) would mean that this press starts on frame 233 and releases at frame 256. Essentially, we will use the four bit encoding and populate the press start and end time for each finger. There may also be the case that the user holds down an input all the way until the end of the video. This would result in a press that never gets released, so we will make sure to taper that loose end by adding a final pair where the end time is the final frame position number. Using the fps numerator and denominator pair we will transform from frame time to millisecond time. This is why the floating point value is important. We can also, if we want, scale presses. For instance, if we multiply the fps float by 1.5, our replay will be 1.5x faster. Our final data structure will be what will be read during the replay time,when the presses are actually played back. This will consist of four vectors, where each vector stores a

pair of hold and release time in milliseconds, that is, how long to hold a press for, and how long to be in a release state. The next element will follow suit, with the next press happening immediately after the previous release time ends. On the last element for each vector, the release time will be 0.

The final phase of this program is the replayer phase. To actually run the presser program, press H on the first note. This presser program trims any empty encodings at the start of the sequence, so the presser program will start on the first pressed key. As soon as you press H, the replaying will begin, and J will terminate it early. The four vectors are passed to four thread routines, along with the associated virtual key. There will be start time padding for any fingers that did not start on the first press, to delay them to the correct time position. We will use <windows.h>'s "keybd_event" function to handle key presses/releases. Using the <chrono> library's time objects we are able to block a thread for however many milliseconds as needed. We first store the start time by getting this current time of execution using "std::chrono::high_resolution_clow::now();". For a thread, we will iterate through our hit vector where first we press a key, then we do "std::this_thread::sleep_for(std::chrono::milliseconds(hit.first));" This is the press time. We will release the key once our thread unblocks, again with "keybd_event". We then sleep until the next key press is ready by adding the last press time with the "hit.first" and "hit.second" integers, this will sleep until the next press start time. Once all threads are complete by either reaching the end or pressing the 'J' key to terminate the threads early, the program terminates. In case you didn't start the program on exactly the correct frame, you can also shift the press/release times by +-3ms by pressing the '1' and '2' keys respectively.

The input arguments into the python program are like so; argv[0] = "InputVideo.mp4" (.mov also works) and argv[1] = "OutputEncoding.txt". Our presser program's arguments are; argv[0] = "OutputEncoding.txt". Our final program was accurate about 95% of the time, tinkering

with the “mp.hands” thresholds and our own threshold y-level shift are what matter the most, as well as a good input video. For a play with a 99% accuracy on a harder map, we got about a 97-98% accuracy. An example of a showcase can be found here (note: the GUI is not part of the CLI program): https://files.catbox.moe/en9s9j.mp4. Don’t forget to set the key bindings in whichever rhythm game you are planning to play.

References

We didn’t use anybody else’s code, we did however reference the documentation for many of the libraries used in this project such as;

- https://docs.opencv.org/4.x/dd/d43/tutorial_py_video_display.html
- https://mediapipe.readthedocs.io/en/latest/solutions/hands.html
- https://docs.python.org/3/library/ctypes.html
- https://en.cppreference.com/w/cpp/chrono
- https://en.cppreference.com/w/cpp/thread/thread
- https://en.cppreference.com/w/cpp/thread/mutex
- https://en.cppreference.com/w/cpp/atomic/atomic
- https://en.cppreference.com/w/cpp/thread/condition_variable
- https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-keybd_event

Results and Improvements

We are overall quite pleased with the results. Even if it’s not a one to one copy of the actual playing, reaching 95% accuracy is still incredibly impressive. To improve the press detection we could tinker a bit more with the press thresholds. From the mp_hands context

manager threshold settings, there's the "min_detection_confidence" and "min_tracking_confidence" thresholds; depending on the video, you can generate different output encodings. You can also tinker with the press-threshold found in the config file. We didn't test this, but setting static_image_mode to true could possibly help, where it runs the process on each frame without regard for the tracking of the hands via previous frames. The biggest advice to get more accurate results is to tinker with the thresholds and have a clear view of your hands. Having more than one video angle to improve the accuracy of the results could potentially be considered, where we have let's say 10 video angles and average out the press encodings, but that's well beyond the scope of this class. A field that uses this concept is the field of 3D photogrammetry, where they use giant camera arrays to generate accurate 3D models based on the photos. The same kind of technique could be used here, except using a set of videos, to get more consistently accurate results.

This program can be easily extended to include more fingers in the process check, in our presentation we extended this program to 7 keys. To achieve the 7 key encoding we had to record a video of the player's thumb as it was obfuscated in the front view. We believe that this concept could be extended beyond rhythm games. A multi-cam setup that showed perspectives from both the front and the top could be used to capture presses across the entire keyboard, as well as being able to determine the key press from the video itself. This could recreate ANY sequence of keyboard inputs, and would be applicable to any kind of game or computer application. Even beyond the computer, the general concept could be used to analyze something such as the playing of a pianist on a piano, for example. If combined with the multiple video angles, you could generate a nearly one to one playback of the pianist's playing through video analysis and a medium like a midi file.

Member Contributions

Ryan Magdaleno led the group on the project idea, had a part in the vision program, wrote the presser program, made and presented a slide, and wrote a decent amount of the implementation section onwards.

Aaron Willming asked great feasibility questions, gave good insights into how things could be implemented, handled the creation of many slides, wrote a large number of the sections at the start of the report, proofread, edited and revised the final report, wrote some portions of the vision program, and also presented a slide.

Semih Kesler helped by writing the first version of the python vision program which was implemented using angle calculations of the landmarks of the fingers which was an approach to detect a press. Helped finish the slides and presented a slide. Added opinions/experiences to some sections of the final report and helped edit/revise the final report.

Jonathan Hung also asked some great feasibility questions, and gave good insights into possible implementations. Jonathan along with Aaron wrote many of the first sections. Jonathan wrote and presented a couple of slides, and wrote some portions of the vision program. Edited and revised the final report.