Image and Video Processing Final Project: OpenCV Duel

Gilberto E. Ruiz, Kyle Maiorana

ELE 400, Fall 2022, Image and Video Processing, Syracuse University
{geruiz@syr.edu}, {kdmaiora@syr.edu}

ABSTRACT

For our ELE 400 final project with Professor Senem Velipasalar, we created a Python-based immersive combat video game that allows a player to interact with a virtual opponent using computer vision. This game comprises of MediaPipe hand tracking, virtual weapons, sound effects, music, and a scoring system that indicates the player's performance.

1. INTRODUCTION

The hallmark of an excellent final project is an engaging and entertaining demonstration that an audience can interact with. To accomplish this for the final project in Image and Video Processing, we wanted to create a real-world variation of one of our favorite interactive video games. There have been several hit implementations of dueling sword-fighting game modes over the years, though they primarily use infrared tracking to capture movements. For this project, we instead used a camera feed to capture a player's real-time hand movements as they hold an imaginary sword. During the game, the player combats a virtual opponent's blade, and their performance is tallied with a running score value that is displayed on the screen. The software overlays a blue line on the screen indicating where the player's real-life sword would be based on their hand coordinates. As the game progresses, the opponent's displayed red sword moves around the screen indicating different attacks being made, which the player needs to counter to prevent a penalty. The locations of both swords are tracked in a data structure, and the software determines at specific intervals if the attack was successful or blocked. The game keeps track of the player's available 'lives', and penalizes them by increasing the speed of the attacks as mistakes are made. This project is fun and playable and takes advantage of powerful computer vision technology that is discussed below.

2. PROPOSED METHODS

2.1. Hand Tracking

The primary technology that drives the functionality of the game is the tracking of the player's hands using their computer's camera. It is necessary to have a dependable tracking model so that the virtual swords and subsequent gameplay are not flickering or inconsistent as the player moves around the screen.

Several methods were explored unsuccessfully for this step of processing but the most effective strategy proved to be the use of a segment of Google's MediaPipe machine learning platform. According to mediapipe.dev, "MediaPipe offers open source cross-platform, customizable ML solutions for live and streaming media." One of these functions is a robust hand-tracking API that uses individual video frames to determine the locations of 21 three-dimensional landmark points. These sites are (X,Y) coordinate representations of key locations on a human hand.

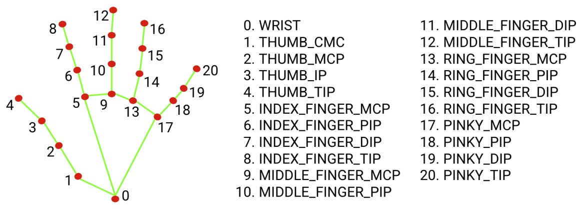
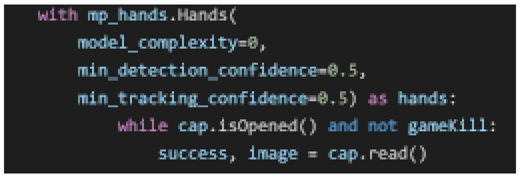


Image: The 21 hand landmarks used by Mediapipe

Mediapipe's hand landmark is trained on a library of over 30,000 real-world hand images human-annotated for use as a sample basis. Since the model processes on a frame-by-frame basis, the key points can be extrapolated at a specific time *t* to influence on-screen behavior and other processing.

The base implementation of the model was accomplished by integrating the starter code provided by Google, mainly the `mp_hands.Hands()` function. This initializes the tracking that can be applied to a frame of a captured video feed. The determined landmark points were stored in an accessible list object called `results.multi_hand_landmarks` which could be referenced.



```
with mp_hands.Hands(
    model_complexity=0,
    min_detection_confidence=0.5,
    min_tracking_confidence=0.5) as hands:
        while cap.isOpened() and not gameKill:
            success, image = cap.read()
```

Image: Initializing of the tracking code on a frame of the video capture

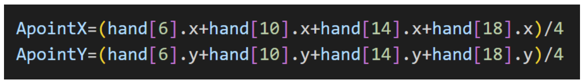
It was necessary for us to configure the model for our purposes by tweaking the parameters. At first, the model was not picking up a hand that was moving around the frame while clenched in a fist. By decreasing the *model_complexity* we eliminated latency issues that were making the game laggy while giving the processor more time to think. By decreasing *min_detection_confidence* from the baseline value of 0.7, the model had more freedom to guess when knuckles and fingers were obscured by the palm. We also experimented with changing the *min_tracking_confidence* values, but 0.5 ended up being a happy medium that led to smooth gameplay.

This tracking was used to develop anchoring locations for the graphics implementation described below.

2.2. OpenCV for Graphics

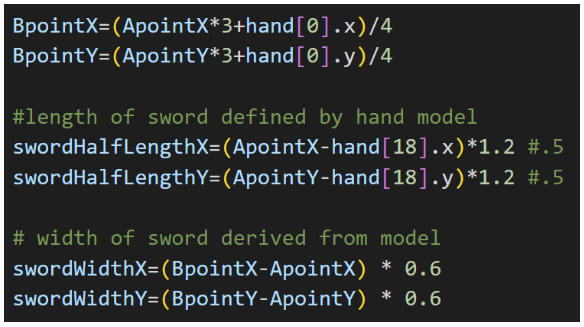
Once a dependable model was capable of picking up a player's hand movements, a digital sword needed to be drawn on the screen. The base for this drawing was the [6], [10], [14], and [18] landmarks shown above. This line across the base of the fingers proved to be the key location for a sword to be extrapolated from. From there, using geometry techniques, the sword could be constructed into a blade and a handle. A metric that needed to be tweaked was the coefficient applied to the sword's length and width to find a feasible relationship that made sense in the coordinate space.

The four anchor points were averaged using the following code:



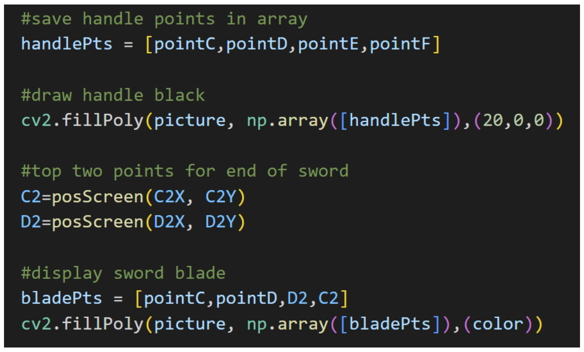
```
ApointX=(hand[6].x+hand[10].x+hand[14].x+hand[18].x)/4
ApointY=(hand[6].y+hand[10].y+hand[14].y+hand[18].y)/4
```

From this, additional anchor points in space could be derived, including a spatially relative length and width:



```
BpointX=(ApointX*3+hand[0].x)/4
BpointY=(ApointY*3+hand[0].y)/4

#length of sword defined by hand model
swordHalfLengthX=(ApointX-hand[18].x)*1.2 #.5
swordHalfLengthY=(ApointY-hand[18].y)*1.2 #.5

# width of sword derived from model
swordWidthX=(BpointX-ApointX) * 0.6
swordWidthY=(BpointY-ApointY) * 0.6
```

By referencing the relationship to the size of the hand, the sword scales according to the distance that the hand is from the camera. Now, we need to extend the appropriate bounds for the handle and blade, store these points in a list, and then display using a CV polygon:

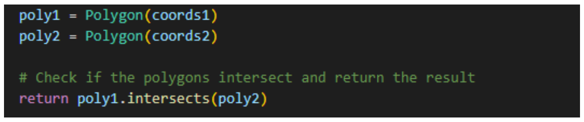


```
#save handle points in array
handlePts = [pointC,pointD,pointE,pointF]

#draw handle black
cv2.fillPoly(picture, np.array([handlePts]),(20,0,0))

#top two points for end of sword
C2=posScreen(C2X, C2Y)
D2=posScreen(D2X, D2Y)

#display sword blade
bladePts = [pointC,pointD,D2,C2]
cv2.fillPoly(picture, np.array([bladePts]),(color))
```

This technique gives a sword blade that appears to be held by the user.

2.3. Shapely Intersection

An important aspect of our game is indicating when the virtual blades have collided. This boolean function determines if the player's score should be incremented, or a life should be lost. Several methods were attempted, including solutions dependent on geometry, slope, and point relationships. The solution that gave the desired result was the Shapely.geometry library, more specifically the Polygon function. From our research, Shapely is a BSD-licensed Python package that can analyze different planar geometric objects which is exactly what we needed for our virtual blades.



```
poly1 = Polygon(coords1)
poly2 = Polygon(coords2)

# Check if the polygons intersect and return the result
return poly1.intersects(poly2)
```

Image: Built-in functions from the Shapely library

The image above demonstrates how we utilized the built-in functions of the Shapely library to account for the intersection aspect. This function works by generating a 2D polygon derived from a passed-in list of coordinate points. The variable *coords1* represents the player's blue sword and *coords2* represents the enemy's current attack (more details about how we created the coordinates for the enemy's attack can be found in section 3.1). With these two arrays of coordinates analyzed and transformed into Polygon objects, the *intersects()* function checks if the objects overlap each other, in other words, if the virtual blades are colliding with each other. We implemented these function calls into a boolean operation that returns True or False values to determine if the blades are colliding or not. This value is then used later on in the game logic.

2.4 Pygame Main Menu Screen

In an effort to make the project seem as close to a traditional video game as possible, the main menu display was implemented. To create this, we utilized an open-source package for Python, Pygame, intended to help structure games and other multimedia applications. For our main menu display, we used Youtuber BaralTech's video and example code as a reference. His simple and visually appealing design of a menu system was ideal for our purposes. It includes three buttons that change color when a mouse is over them and a quit button that closes the function.

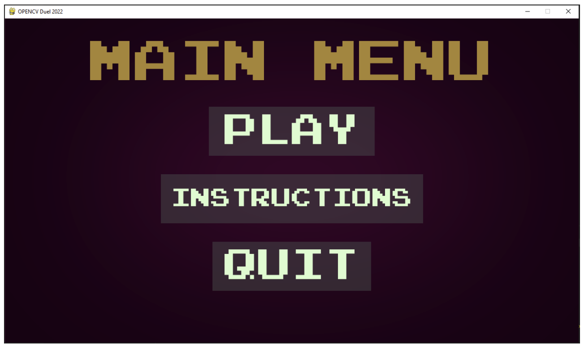
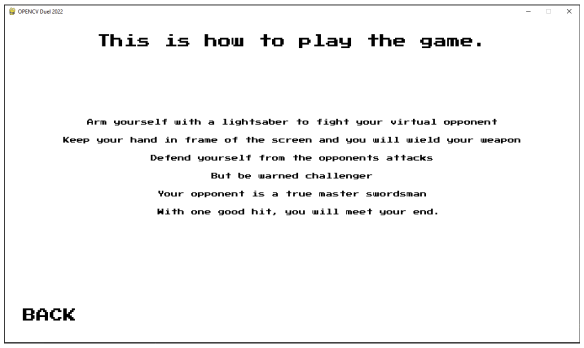


Image: First page of the Main Menu



OPENCV Duel 2022

This is how to play the game.

Arm yourself with a lightsaber to fight your virtual opponent
Keep your hand in frame of the screen and you will wield your weapon
Defend yourself from the opponents attacks
But be warned challenger
Your opponent is a true master swordsman
With one good hit, you will meet your end.

BACK

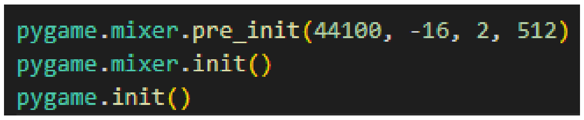
Image: Instructions page of the Main Menu

We took advantage of the extra screen to create an instructions page as a way to explain in text what the user needs to do for the game. This function should hopefully eliminate unnecessary confusion and frustration for the user.

2.5 Pygame mixer for audio implementation

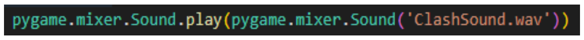
To further immerse the user into a video game environment, we utilized the Pygame library's built-in "Mixer" functions in order to play music/audio files in our Python script. The *Pygame.mixer* module contains classes for loading sound objects and controlling playback. For our program, we created a total of 4 audio/music files. The files consist of two background songs (MainMenuTheme.wav and FightTheme.wav) and two short sound effects (ClashSound.wav and DeathSound.wav). The first background song known as MainMenuTheme is only utilized at the beginning of the code. When the user is at the main menu screen, it'll play a well-known piece of music from the Star Wars franchise called Jedi Temple March. Once the user decides to press the play button on the main menu screen, MainMenuTheme.wav will stop playing and the code will call the other .wav file. For OpenCV Duel, we included an iconic score example from the Star Wars franchise, Duel of Fates to set the ambiance. For the last two short sound files, we programmed these to trigger once the user has blocked, and missed the opponent's attacks respectively. The sounds consist of a lightsaber clash, as well as a scream that lets the user know that you've been hurt and lost a life in the game. The explanation of how we acquired these sound files will be further explained in section 3.2.

With the files loaded into the code repository, we now had to utilize the built-in Pygame functions as shown below:



```
pygame.mixer.pre_init(44100, -16, 2, 512)
pygame.mixer.init()
pygame.init()
```

Image: pygame.mixer functions to initialize the process



```
pygame.mixer.Sound.play(pygame.mixer.Sound('ClashSound.wav'))
```

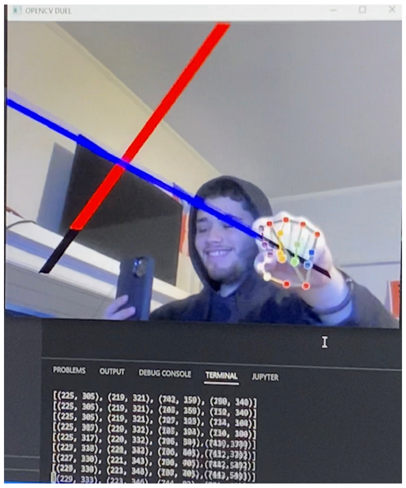
Image: pygame.mixer function to load the sound file

The first image above is the process to initialize the Pygame.mixer functions, and the second image is the actual use of these functions used later on in the program.

3. EXPERIMENTS

3.1. Coordinates for attacks

With the player's hand being successfully tracked and having a virtual sword placed perfectly on reference points 6, 10, 14, and 18 out of the 21 provided, an opponent needs to be designed. We configured print statements to display the live coordinates of the user blade, and used this information to come up with different attack patterns for the video game opponent. To accomplish this we borrowed a technique from video game and movie production, motion capture. Since we had a way to translate real-life movements into computerized coordinate data, we could then use our bodies to explore the relationship between the screen and 3D space. We then needed to record different coordinates from the screen to capture and store in a data structure of attacks that work strategically for the opponent. These points could then be used to draw the red blade, and have the villain fight the player.



```
[(225, 305), (219, 321), (202, 159), (200, 340)]
[(225, 305), (219, 321), ([illegible], 359), ([illegible], 349)]
[(225, 305), (219, 321), ([illegible], [illegible]), ([illegible], [illegible])]
[(225, 307), (220, 322), ([illegible], [illegible]), ([illegible], [illegible])]
[(225, 317), (220, 332), ([illegible], [illegible]),([illegible],[illegible])]
[(229, 338), (220, 342), ([illegible], [illegible]),([illegible],[illegible])]
[(227, 330), (221, 343), ([illegible], [illegible]),([illegible],[illegible])]
[(228, 330), (221, 348), ([illegible], [illegible]),([illegible],[illegible])]
[(229, 333), (223, 346), [illegible]
```

Image: Blue Blade's coordinates being printed for recording

The image above demonstrates a live recording of the blue virtual sword being continuously tracked and aligned with the user's hand. In that same loop, the sword's blade coordinates are printed out in the VSCode terminal below for reference. A working method we found for capturing these coordinates was to orient the user's hand to the desired location and stay perfectly still while using the other hand to obscure the camera lens. This paused the code from looping and freezing with the very last coordinate printed at the terminal. This experiment worked effectively because of the code's design feature where if no hand is detected, the raw camera feed is shown so the user can re-center themselves. If the video feed is currently tracking a hand and is suddenly blocked, it can't track the hand anymore, and the virtual sword disappears.

At this point, the looping segment of the code is paused. We used that detail to our advantage and acquired a total of seven different attacks for our game. The image below represents all of the attacks printed onto the screen at the same time to represent the variety of attacks to be found in the game.

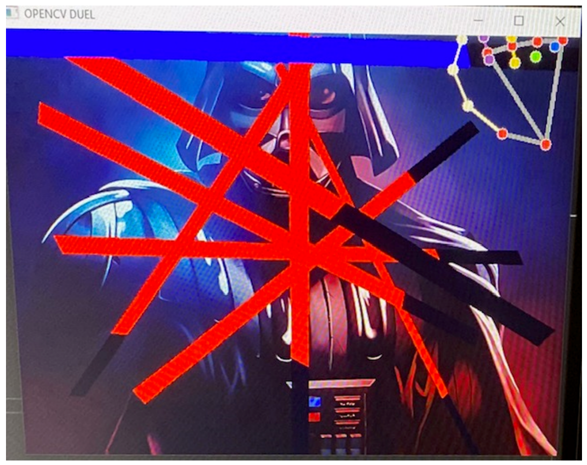


Image: Every possible attack for OpenCV duel printed together

3.2 Retrieving Audio Files

The Pygame mixer was used to load different sound files into the game. To acquire these files, we searched on Youtube for video files that had the desired songs and sounds. The URL could then be passed to a Youtube-to-mp3 file converter tool. At first, we utilized a website tool,Y2Mate, to convert the Youtube video into a MP3 file. As we tried to load the MP3 files into the code, some of our computers would not recognize the file when using the Pygame.mixer built-in functions. To fix this problem, we restarted the process using a Youtube-to-Wav converter tool called Motionbox. With the audio files now converted to wav files, our computers had no issue recognizing them anymore.

CONCLUSION

Throughout the process of designing and implementing this game, we had the opportunity to practice a number of new skills. Increasing our familiarity with image manipulation techniques and expanding our experience with machine learning processing methods were both products of this exercise.

We hope that the final result is a fun way to experiment with an application of image and video processing, and is an enjoyable experience for the user.

REFERENCES

https://mediapipe.dev/

https://google.github.io/mediapipe/solutions/hands

https://motionbox.io/tools/youtube-to-wav

https://en.y2mate.is/125/youtube-to-mp3.html

https://www.geeksforgeeks.org/python-playing-audio-file-in-pygame/#:~:text=In%20order%20to%20play%20music,Sound%20objects%20and%20controlling%20playback.

https://github.com/baraltech/Menu-System-PyGame