Niv Sherf, Raz Yehuda, Keren Mazaki, Roi Mozes

Infinity Blade

Playing a mobile game with gestures.

"Infinity Blade" is an action RPG game crafted on the Unreal Engine. Within the game, players engage in one-on-one battles against a variety of adversaries. In the original gameplay, players utilize swipe gestures to execute attacks and parries, while tapping the screen facilitates dodging and blocking enemy assaults. In our project, we've employed classical image processing techniques, allowing players to control the in-game character using a camera and a plastic sword.

In our project we wanted a fast, reliable way to convert human gestures into specific inputs for our game. The most important thing is that the game would act in a real time manner, being fast, responsive, and precise to allow the player to correctly handle the already quite fast paced game without being bogged down by the delay from the translation layer.

In our game we track the movement of the player to the left or to the right to determine dodging direction, using a background subtraction mask and contour detection. We track the sword using an HSV mask and a Kalman filter. We also find when the player is blocking by raising the sward above their head, with a Hough transform to find a straight line withing the HSV color mask of our green blade, in a specific area of the frame.

Originally, we wanted to use some sort of ML algorithm such as "Haar Cascade" to recognize the sword and player, then track the movement of the various elements and add in-game logic for robust tracking. The issue with that approach was that the variety of ML algorithms, while quick in the classical sense, are very slow for real-time applications where every frame counts, and therefore sometimes we need to anticipate the movement of the player before they actually perform it. That led us to use the Kalman filter algorithm for tracking an object. And background subtraction with segmentation for detection of the major movement axis. The benefit of both of those algorithms is that they are extremely quick and allow for precise input without significant delay, while only adding the constraint that the background would be clear of significant movements that will overshadow those of our player.

The camera was positioned directly in front of the player, centered within the frame. Deviations occurred when the player stood very far or very close to the camera, making precise tracking challenging within our set threshold for tracking angles and distances.

The camera we chose to use is the laptop camera. The game worked for every camera; however, we found that less color calibration is required when we use a persistent camera.

The main challenges of the project are as follows:

- Tracking the sword robustly and quickly.
- Determining movement direction quickly, without confusing the movement of the player with the movement of the sword.
- Finding an intuitive way to block within the game, as we didn’t use a shield, so we needed to find a creative way to gesture blocking.

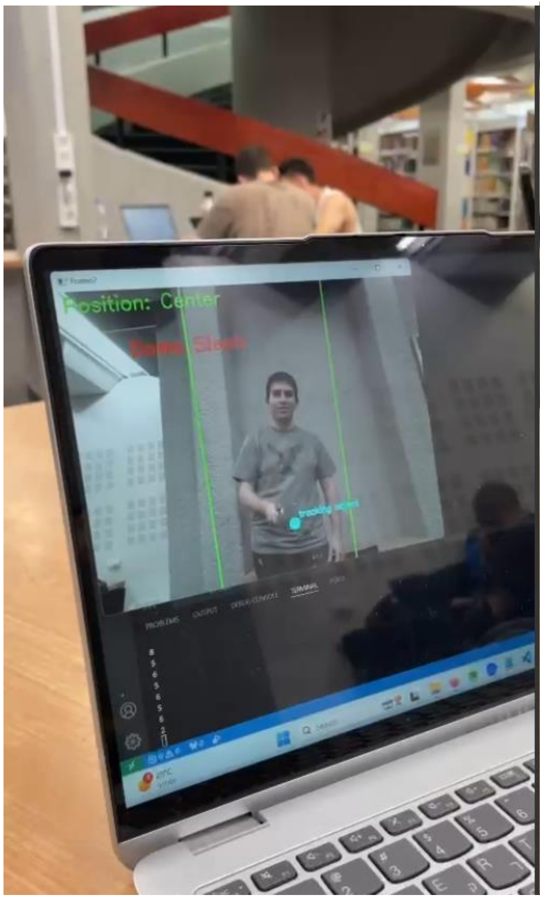
Some constraints:

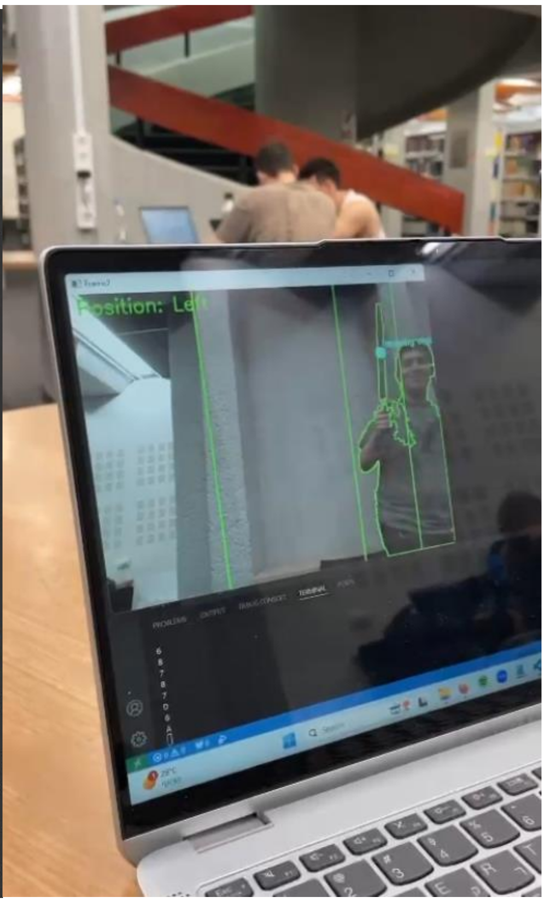
- The game requires a mainly static background, which follows from the background segmentation algorithm used. Despite the segmentation into various types of movements, we still found that a large enough movement will confuse the system responsible for dodging and create erratic results.
- Using ML algorithms to recognize the sword regardless of color proved too slow. Therefore, we painted the sword in a particular color so that we would be able to use a color mask.
- That raised the issue of the players wearing the precise type of color as our sword, causing the system to be confused. While we did use a very precise range of colors converted to HSV to mitigate this issue, we found it prudent to keep an extra sword in a different color to allow players that were wearing a clothing item with the exact color of the sword. That proved redundant as even our green-wearing players experienced no issues with the tracking of the sword, since the HSV mask was very precise.

Some assumptions:

- We assumed the background of the game remains roughly static.
- We assumed the lighting of the location remained mostly constant, accounting for light and shadow created from movement (We kept our color filter very precise).

The system in action:





- Player position in the center
- Slashing down

- Player position in the left
- Not slashing

Let's categorize the algorithms utilized in the project into three sections: attacking, blocking, and dodging.

Attacking: When we wanted to detect the movement of the sword, we used color-based detection and painted the two swords in 2 different colors: green and purple. Then we applied a color mask to detect the specific color of the sword on the screen in order to track the location of the sword and distinguish it from other movements on the screen.

The color mask was based on the VHS color-space. We established lower and upper thresholds for each channel to identify the precise color of each sword.

Now, when we can track the movement of the object, we want to predict in which direction it is moving. for that we used Kalman Filter.

The Kalman filter is a clever tool used in many fields to figure out what's happening in a system that's changing over time. Imagine you're tracking a moving object, but the measurements you're getting are a bit fuzzy due to errors. The Kalman filter steps in to help by combining what it thinks the object's path should be, based on its past behavior, with the fuzzy measurements it's getting right now. By blending these predictions with the real measurements, it gives us the best guess about where the object actually is.

In simpler terms, the Kalman filter is like a detective trying to piece together a puzzle. It uses clues from the past and new evidence to update its understanding of the situation. Even if some of the evidence is unreliable, it still manages to come up with a pretty accurate picture of what's going on.

When it comes to image processing, the Kalman filter can be handy too. Let's say we're trying to clean up a blurry image. Just like with tracking objects, the filter uses what it knows about how images change over time to sharpen up the blurry parts. It's all about refining our understanding of the image based on both our expectations and the actual data we have.

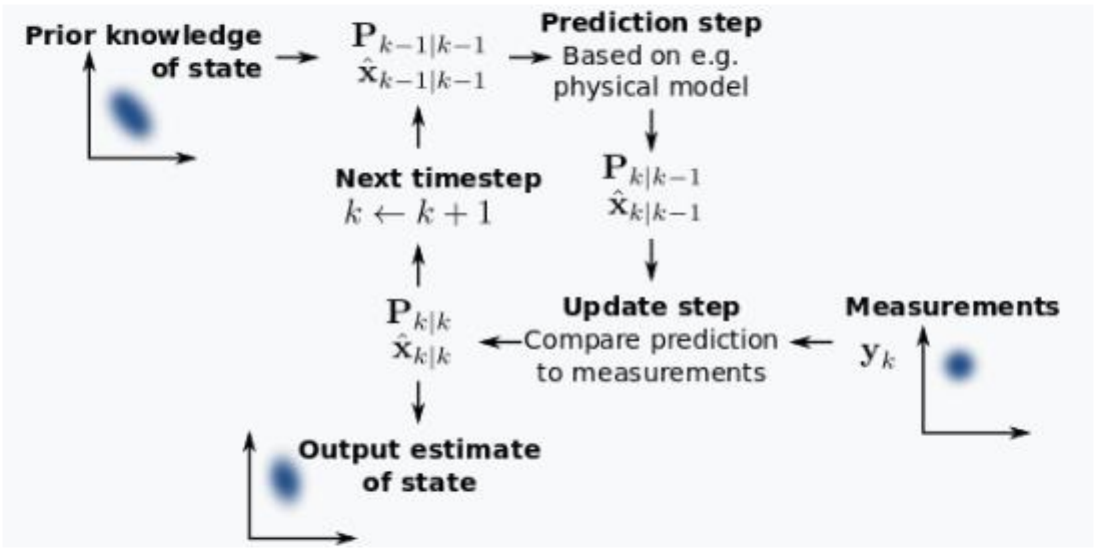
Now, the Kalman filter isn't magic—it has some rules it follows. It assumes that the system it's dealing with behaves in a predictable, linear way. It also assumes that the errors in our measurements follow a particular pattern. As long as these assumptions hold true, the filter can work its magic and give us reliable estimates.

In terms of what we feed into the Kalman filter, it's pretty straight forward. We start with an initial guess about the system's state and how uncertain we are about it. Then, we give it the measurements we're getting over time, along with some details about how the system behaves and how noisy our measurements are.

The output of the Kalman filter is an updated estimate of the system's state, along with a measure of how certain we are about that estimate. It's like getting a clearer picture of where the object is or what the image should look like, along with a confidence level attached to that information.

To make the Kalman filter work for different scenarios, we can tweak some parameters, like how the system behaves or how noisy our measurements are. But no matter the details, the filter follows the same basic steps: predict, update, and repeat. It's a reliable tool for making sense of changing systems, whether we're tracking objects or cleaning up images.

We can view how the Kalman filter works in the next flowchart:


Prior knowledge of state
$\mathbf{P}_{k-1|k-1}$ $\hat{\mathbf{x}}_{k-1|k-1}$
Prediction step
Based on e.g. physical model
Next timestep
$k \leftarrow k+1$
$\mathbf{P}_{k|k-1}$ $\hat{\mathbf{x}}_{k|k-1}$
$\mathbf{P}_{k|k}$ $\hat{\mathbf{x}}_{k|k}$
Update step
Compare prediction to measurements
Measurements
$\mathbf{y}_k$
Output estimate of state

The algorithm receives an initial state estimate: $\hat{x}_{0|0}$ and covariance: $P_{0|0}$

next step is to predict the next state estimate and covariance:

- $\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k$
- $P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k$

Where: F_k is the state transition matrix, B_k is control input matrix, H_k is the measurement matrix and Q_k is the process noise covariance matrix.

Now we update the state estimate and covariance according to the Kalman gain:

- $K_k = P_{k|k-1} H_k^T \left(H_k P_{k|k-1} H_k^T + R_k\right)^{-1}$
- $\hat{x}_{k|k-1} + K_k(z_k - H_k \hat{x}_{k|k-1})$
- $P_k = (I - K_k H_k) P_{k|k-1}$

Then we use the updated step for the next prediction.

Blocking: To trigger the blocking command, the player must hold the sword horizontally above their head and maintain this position for a few seconds. Our algorithm for recognizing blocking utilizes the Hough lines transform. Initially, we process each frame by applying the HSV color mask used for sword tracking. This step allows us to isolate lines corresponding only to the sword, and not to other elements in the frame. Then we applied the Canny edge detector, which highlights areas with significant intensity changes, effectively identifying edges. Lastly, the edged frame is passed through a Hough transform algorithm, which calculates the slope and intercept for every pixel. Pixels forming a line in the frame correspond to clustered dots in the Hough space, where each dot represents a line, and vice versa. The algorithm then outputs the k most voted dots in the Hough space. After isolating lines in the top half of the frame withing our color mask, we searched for lines within a specific angle range, indicating a horizontally held object.

Several parameters within the algorithm are adjustable. Firstly, the thresholds of the Canny edge detector can be modified, consisting of a low and a high threshold, defining the minimum and maximum intensity gradient values. We set these thresholds to 20 and 500 respectively. Additionally, the kernel size parameter determines the size of the kernel used for computing gradients. In our case, the kernel size was 3x3. Other changeable parameters are:

- ρ- the distance resolution of the accumulator array in pixels. in our project: 1.
- Θ- the angular resolution of the accumulator array. In our project: $\theta = \frac{\pi}{2}$.
- Threshold- the minimum number of votes required for a line to be detected. In our project: 50.
- MinLineLength- specifies the minimum length of a line in pixels, that to not detect small and insignificant background lines. In our project: 60.
- MaxLineGap- indicates whether several lines are considered as one line or more. In our project: 40.

After adjusting the parameters to only detect the sword in the certain location and in the certain slope, we wanted to determine that the player is blocking only if they raise the sword for a few seconds.

We initialized two variables: one named "blocking" and the other named "block."

The "blocking" variable is set to True when a blocking event was detected in the last frame, and to False otherwise. If "blocking" is True, we check for blocking every frame. Otherwise, we only check for blocking every 5 frames to reduce redundancy.

The “block” variable increases linearly each time the blocking detection function is triggered (determined by the “blocking” variable defined above), and a blocking event is detected. It decreases exponentially when there is no blocking detection. Only when this variable reaches a certain threshold, we declare that the player is blocking.

This ensures that the player must maintain the sword in the required position for several consecutive frames for the code to recognize it as blocking.

Dodging: Let’s go over the steps we took in this part, and then explain each image processing algorithm that was used.

First, we subtracted the sward from the frame, using the mask that was applied for the sward tracking.

Following that, we applied background subtraction to the frame, and removed noise. Next, we searched for contours in the foreground image, selecting the contour with the largest area, provided its area surpassed a certain threshold. otherwise, we skipped the frame.

Next, we wanted to determine the contour’s center of mass, focusing on its x-coordinate. To ensure smooth movement of the center across frames, we implemented a smoothing technique using a weighted average of the previous 7 frames, with greater weight assigned to more recent values.

For detecting dodging maneuvers, we established left and right thresholds. If the center exceeded the left threshold, we inferred a dodge to the left; conversely, if it fell below the right threshold, we assumed a dodge to the right.

Let's go through the algorithms that we used:

<u>open CV's createBackgroundSubtractorMOG2():</u>

this Class uses the GMM algorithm, that is used to separate moving objects from the background. In our case, to separate the player from the background. Here's how the algorithm works:

The value of a pixel at time t in RGB or some other color- space is denoted by $\vec{x}^{(t)}$. Pixel-based background subtraction involves decision if the pixel belongs to background (BG) or to the foreground (FG).

given a time period T, we observe each pixel in the frame and look at its last T samples. So at time t, we have a data set $\mathrm{X_T} = \left\{\vec{x}^{(\mathrm{t})}, \dots, \vec{x}^{(\mathrm{t-T})}\right\}$.

For each new sample we update the training data set X_T and estimate $p(\vec{x}^{(\mathrm{t})}|X_T, BG, FG)$. To estimate $p(\vec{x}^{(\mathrm{t})}|X_T, BG, FG)$ we use GMM with M components:

$p\left(\vec{x}^{(\mathrm{t+1})}\middle|X_T, BG, FG\right) = \sum_{m=1}^{M} \omega_m N(\vec{x}^{(\mathrm{t})}; \mu_m, \sigma_m^2 I)$. Where $N(\vec{x}^{(\mathrm{t})}; \mu_m, \sigma_m^2 I)$ is the m'th Gaussian distribution of $\vec{x}^{(\mathrm{t})}$out of M estimated Gaussian distributions, μ_m are the estimates of the means and σ_m are the estimates of the variances that describe the Gaussian components. The covariance matrices are assumed to be diagonal and the identity matrix I has proper dimensions. The mixing weights denoted by ω_m are non-negative and add up to one.

We update the parameters using the following equations:

$$\omega_m \leftarrow \omega_m + \alpha(o_m^{(t)} - \omega_m)$$

$$\vec{\mu}_m \leftarrow \vec{\mu}_m + o_m^{(t)}\left(\frac{\alpha}{\omega_m}\right)\vec{\delta}_m$$

$$\sigma_m^2 \leftarrow \sigma_m^2 + o_m^{(t)}\left(\frac{\alpha}{\omega_m}\right)\left(\vec{\delta}_m^T\vec{\delta}_m - \sigma_m^2\right)$$

where $\vec{\delta}_m = \vec{x}^{(\mathrm{t})} - \vec{\mu}_m$, and α describes an exponentially decaying envelope that is used to limit the influence of the old data. Usually, $\alpha = \frac{1}{T}$.

$o_m^{(t)}$ is set to 1 for the 'close' component with largest ω_m and the others are set to zero. We define that a sample is 'close' to a component if the Mahalanobis distance from the component is for example less than three standard deviations. The squared distance is calculated as: $D^2(\vec{x}^{(\mathrm{t})}) = \vec{\delta}_m^T \vec{\delta}_m / \sigma_m^2$.

If there are no 'close' components a new component is generated with $\omega_{M+1} = \alpha$, $\vec{\mu}_{M+1} = \vec{x}^{(\mathrm{t})}$ and $\sigma_{M+1} = \sigma_0$, where σ_0 is some initial variance. If the maximum number of components is reached, we discard the component with smallest ω_m.

To use the algorithm, we create an object of the class, and then use the "apply" method to a given frame.

The Classes' input parameters are:

- History: the number of last frames to use as the data set. The default value is set to 500.
- varThreshold: the Threshold for determining if a pixel is part of the background. The default value is set to 16.
- detectShadows: If true, the algorithm will detect shadows and mark them. The default value is set to False.

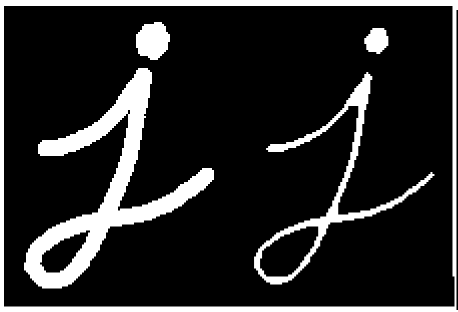
The "apply" function's input is an image in the range of [0,255], and it's output is an 8-bit binary mask.
The function has one parameter- learningRate, which is a number between 0 and 1 that indicates how fast the background model is learnt. 0 means that the background model is not updated at all, 1 means that the background model is completely reinitialized from the last frame.

<u>Open CV's morphologyEx() using the OPENING operation:</u>

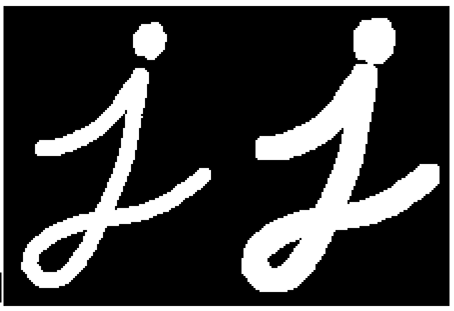
This function is used to denoise an image, in our case, we used it to denoise the frame after applying the background subtraction mask. This function is based on two main operations:

1. Erosion: this operation erodes away the boundaries of the foreground object, by applying a 2D convolution of the image with a kernel in this manner: a pixel is set to 1 (white) only if all the pixels under the kernel are one, otherwise, it is set to zero (black).
2. Dilation: The opposite of erosion. here, a pixel is set to zero only if all the pixels under the kernel are zero, otherwise, it is set to one.

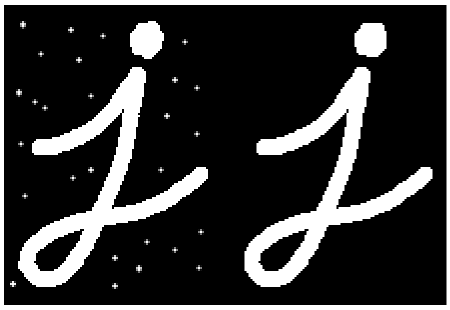
We can combine these two operations in different ways. Here, we chose to use the "Opening" operations, which is applying dilation and then erosion to the image. This way, we denoise the image.



Erosion



Dilation



Opening

The function's inputs are:

required arguments:

- InputArray: the input image, usually a binary image but not necessarily
- op: the operation to apply (dilation, erosion, or one of the combinations)
- kernel: the kernel that is being used

optional arguments:

- anchor: Anchor position with the kernel. Negative values mean that the anchor is at the kernel center. Default value is set to be Point(-1,-1)
- iterations: Number of times erosion and dilation are applied. Default value is set to 1.
- borderType: pixel extrapolation method. Refers to how pixels outside the boundaries of an image are handled. The default value is set to BORDER_CONSTANT.
- borderValue: border value in case of a constant border. The default value is morphologyDefaultBorderValue().

The function's output is an image of the same size as the input image.

<u>Open CV's findCountors():</u>

This function finds contours in a binary image. In our case, we apply it on the masked frame that contains the foreground.

Firstly, let's define these following terms:

- F: the scanned image with indices {fij}.
- NBD (Next Border Number): a sequential number assigned to each border detected in the image. It keeps track of the current border being processed and increments whenever a new border is encountered.
- LNBD (Last Next Border Number on the Current Row): represents the sequential number of the last border encountered on the current row during scanning. It is updated whenever a new border is detected, ensuring proper hierarchy determination for subsequent borders.

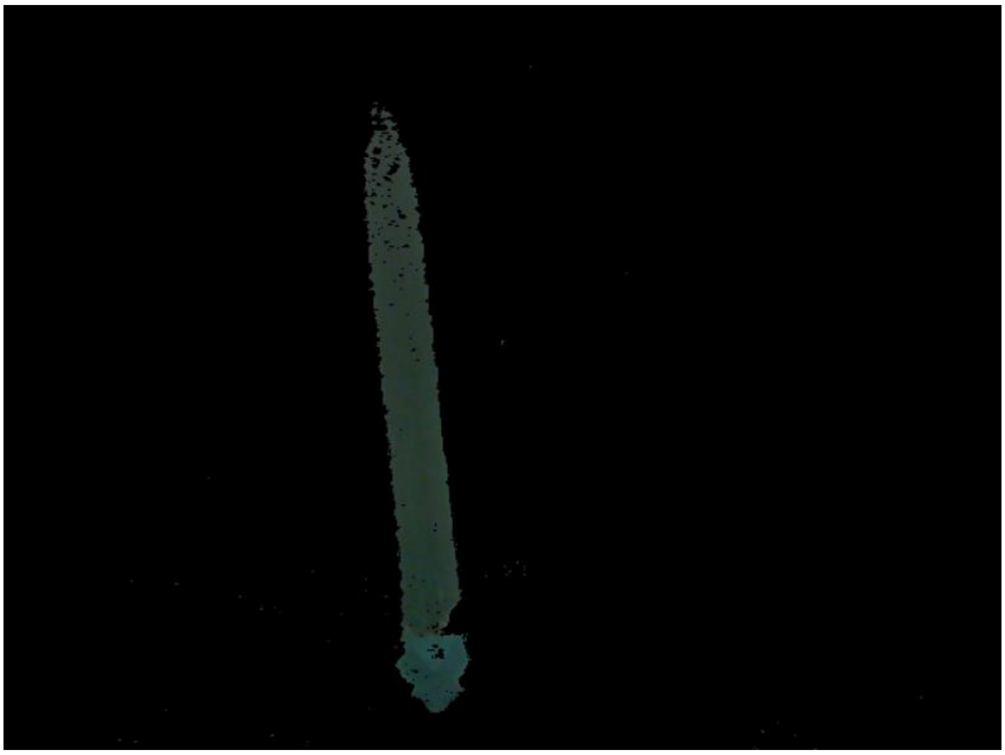
The algorithm works as follows:

1. Initialize NBD to 1.
2. Scan the picture with a top-to-bottom, left-to-right raster.
3. For each pixel (i, j) with $f_{ij} \neq 0$:
    a. Reset LNBD to 1 at the start of each row.
    b. Select one of the following:
        i. If $f_{ij} = 1$ and $f_{ij-1} = 0$, then mark (i, j) as the starting point of an outer border. Increment NBD and set $(i_2, j_2) = (i, j - 1)$.
        ii. Else if $f_{ij} \geq 1$ and $f_{ij+1} = 0$, then mark (i, j) as the starting point of a hole border. Increment NBD, set $(i_2, j_2) = (i, j + 1)$, and LNBD= f_{ij} (if $f_{ij} > 1$).
    c. Otherwise, continue scanning.
4. Depending on the types of the newly found border and the border with the sequential number LNBD, determine the parent of the current border according to the table down below.
5. Follow the detected border: Starting from (i_2, j_2), follow the contour in a clockwise direction until returning to the starting point:
    a. Find the next nonzero pixel in the neighbourhood of (i, j).
    b. Update the current position and continue until returning to the starting point.
6. If $f_{ij} \neq 1$, then LNBD = $|f_{ij}|$ and resume scanning from the next pixel $(i, j + 1)$.
7. Terminate the algorithm when reaching the lower right corner of the picture.

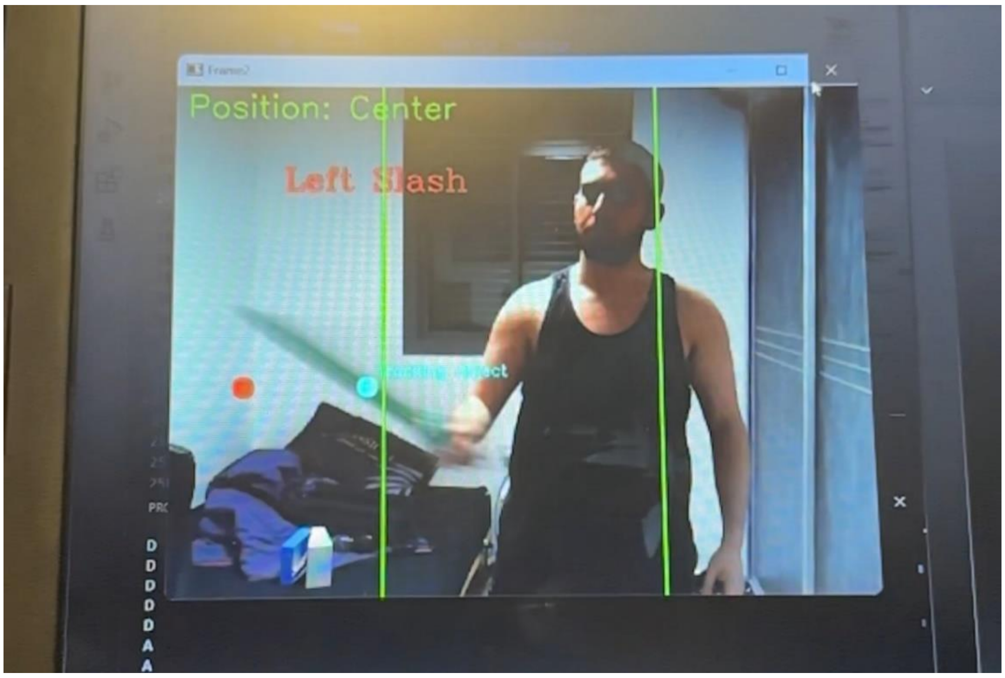
Decision Rule for the Parent Border of the Newly Found Border B:

	Type of border B’ with LNBD	Outer border	Hole border
Type of B			
Outer border		the parent border of the border B’	The border B’
Hole border		The border B’	the parent border of the border B’

Attacking: We used HSV color space to separate the sword from everything else in the background, we can see how the mask we use shows only the sword:



We can see how the Kalman filter predicts the location of the sword for the next frame:

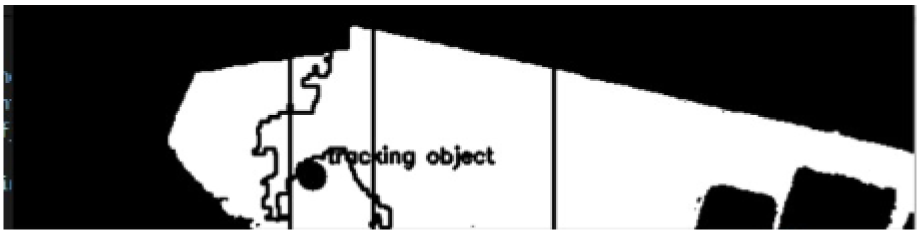


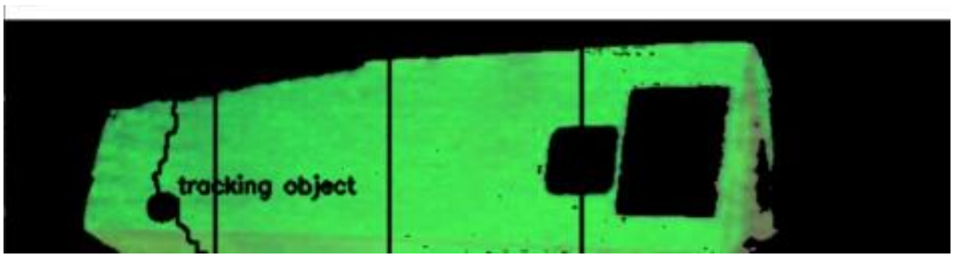
In blue we see the current location of the sword (the center of the sword) and in red the predicted location over the next frame.

Blocking: When we tried to write the code for the blocking function, we had several ideas. The first thing we tried to do is use Haar Cascade, a machine learning algorithm that can identify objects in a frame. We wanted to block if the player raises their hand as a stop sign, next to their face. We first wanted to ask the player if they are right-handed or left-handed, and according to their answer, decide which hand to detect. We assumed that the player would swing the sword with their dominant hand, and use the other hand to block. Then, using Haar Cascade, we looked for a palm recognition in a certain part of the frame, and when we found one, a blocking flag would be raised. That algorithm worked perfectly, but when we combined all the different functions, we found that it ran too slowly, which is unfortunate, since the game should work in real time. After that, we decided to switch to the algorithm described in section 5.

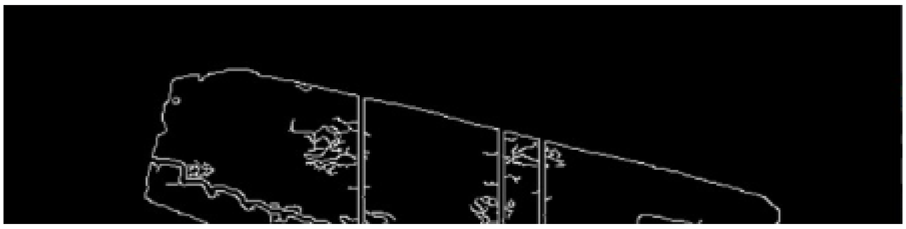
When writing the blocking algorithm, we needed to go through several stages:

At first, we masked the frame so that it keeps track on green objects (the color of the sword) on the top half of the frame. (if it finds more than one green object, it tracks the one with the color that's more like the sword's color).

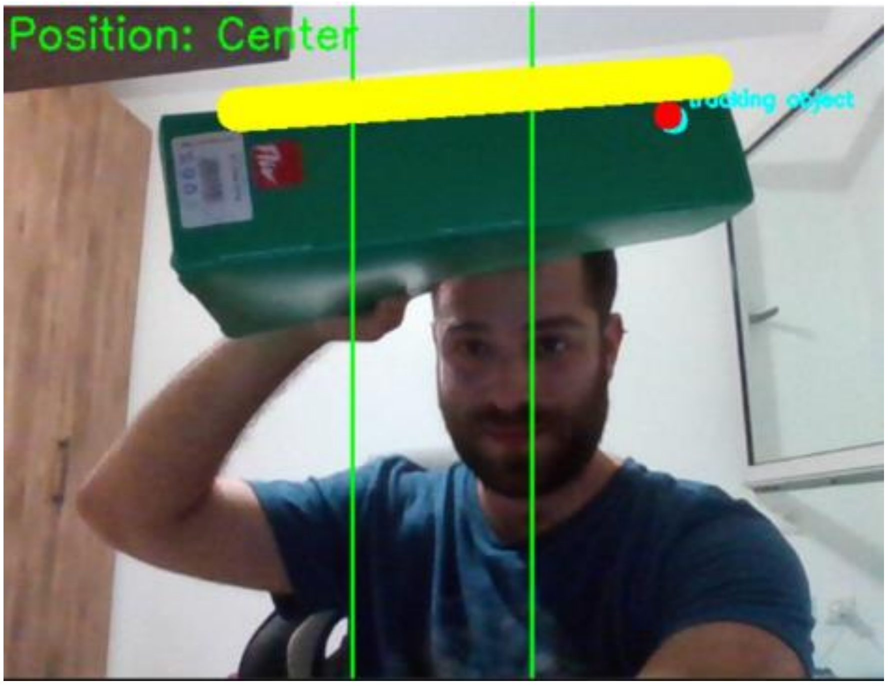




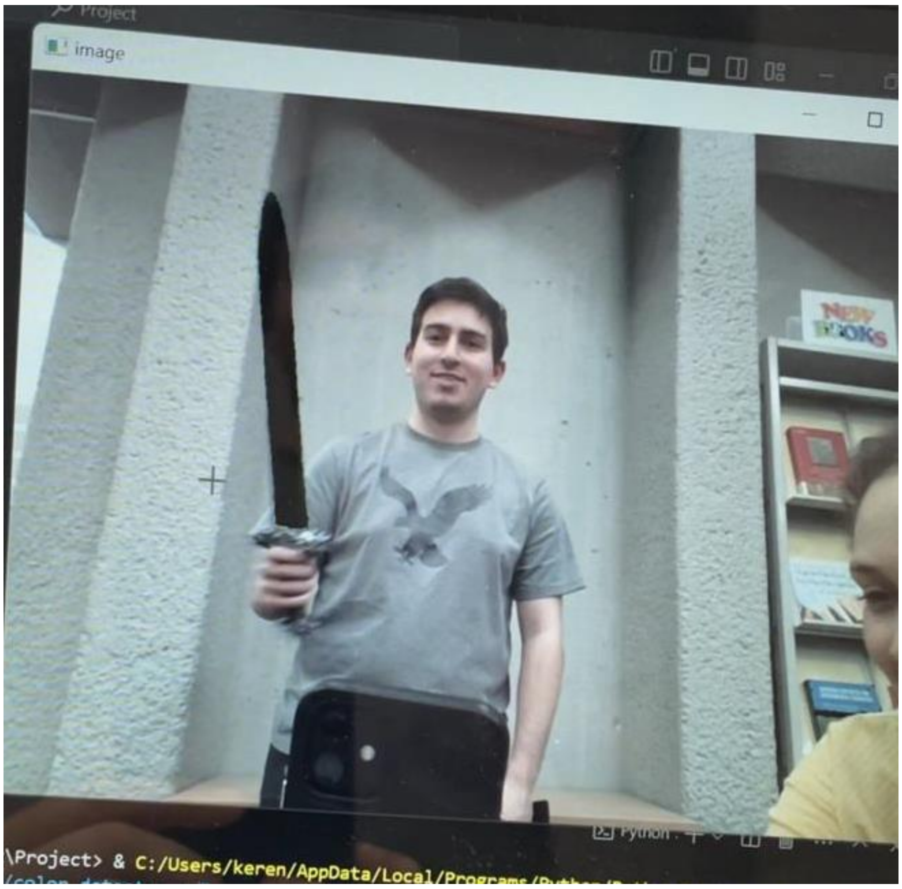
Then, we applied the Canny edge detection algorithm:



Finally, after applying the Hough transform on the Canny frame, it detects a straight horizontal line above the head as blocking:



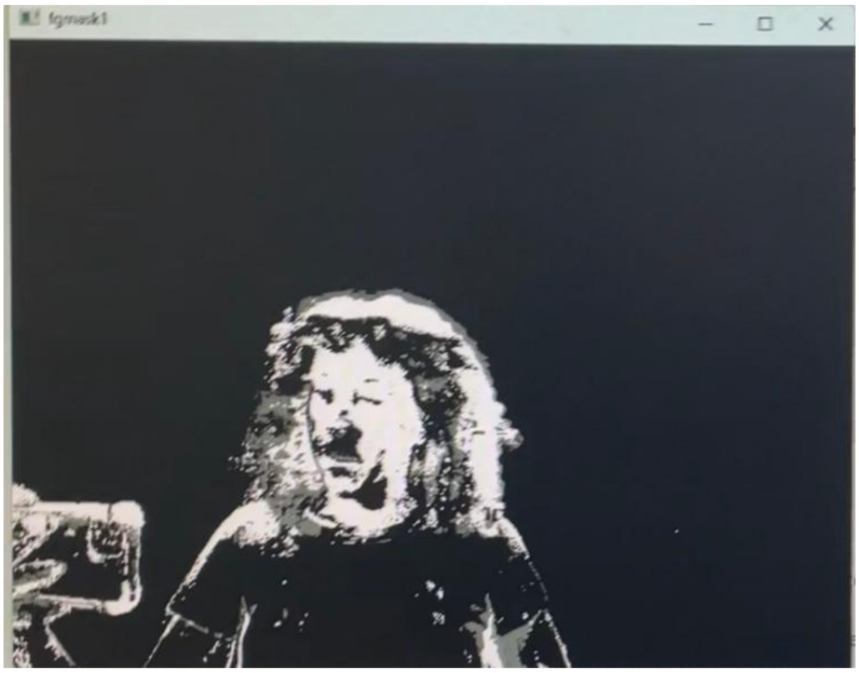
Dodging: sward removed from the frame:



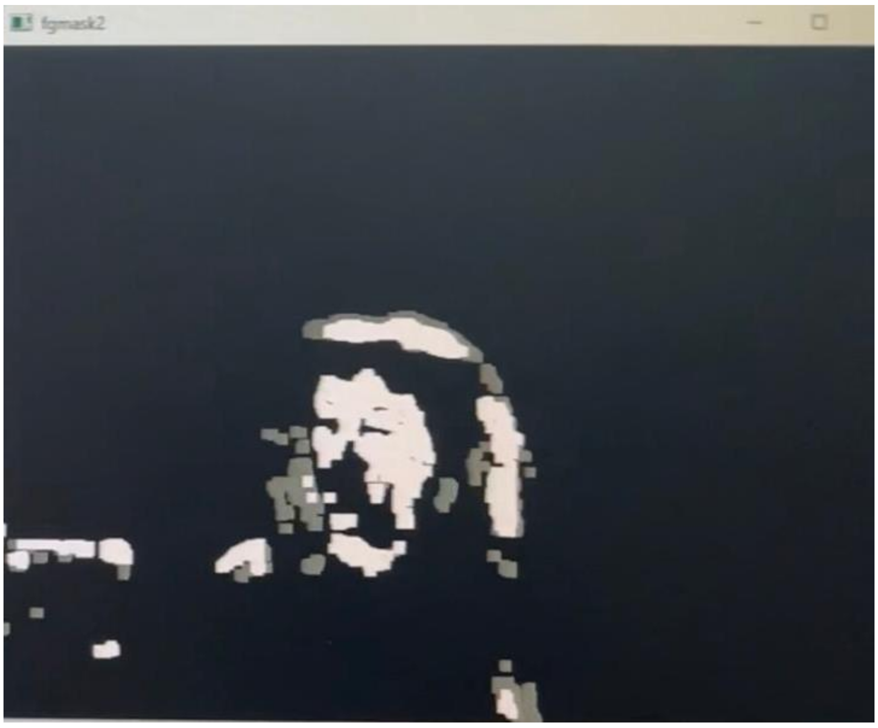
```
\Project> & C:/Users/keren/AppData/Local/Programs/Pytho
```

background subtraction mask:

Before denoising:

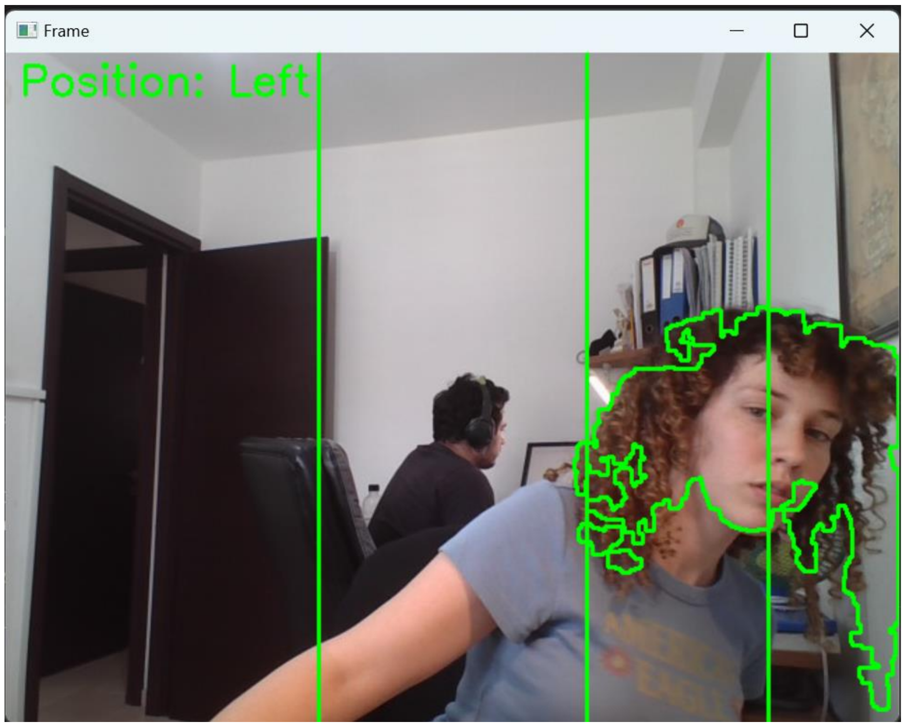


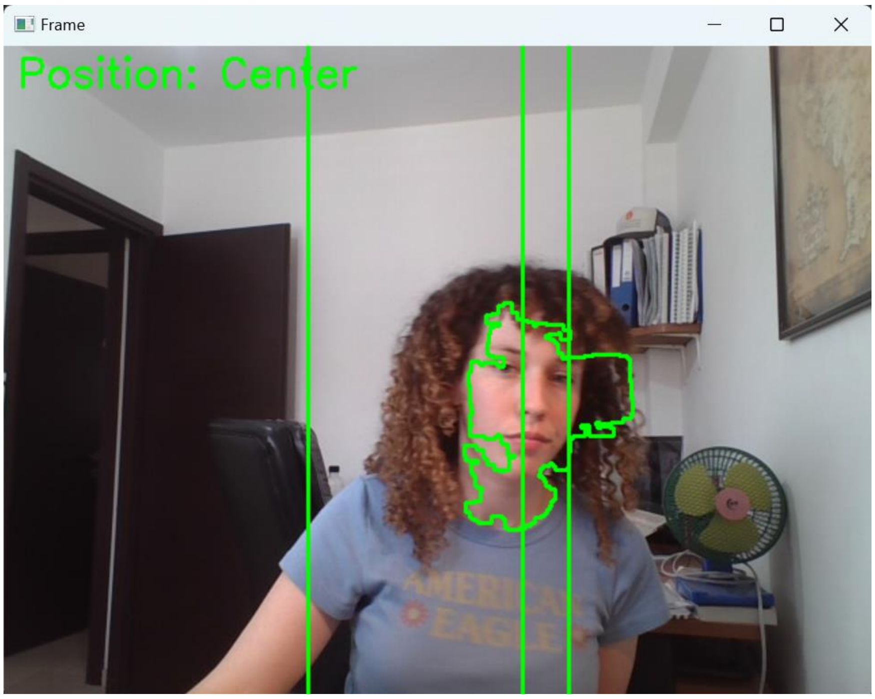
After denoising:

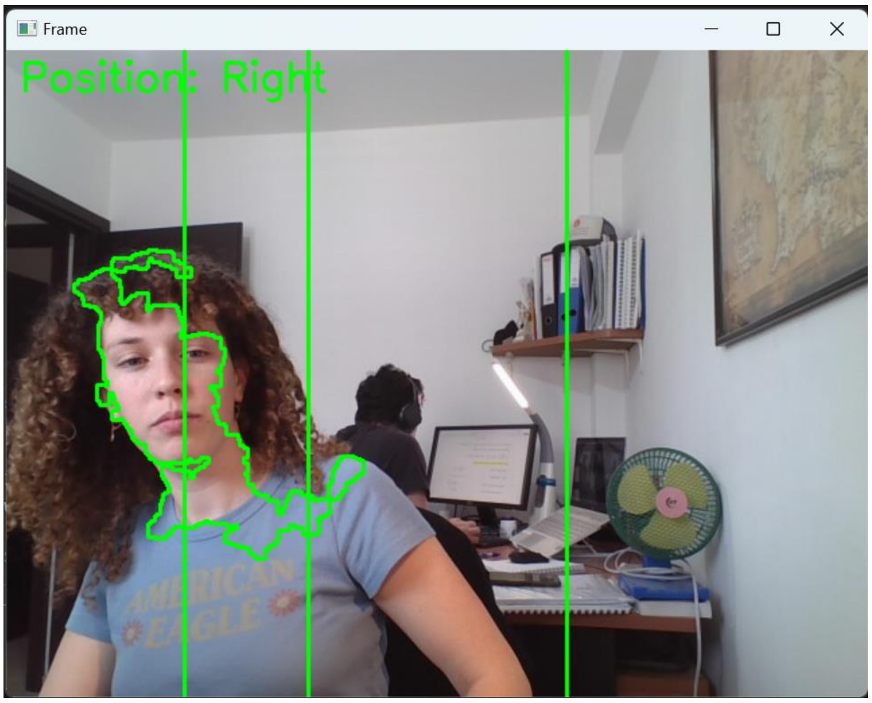


Max contour and right/left dodging detection:

The lines represent the left and right thresholds, and the x coordinate of the center of mass.




Frame
Position: Center


Frame
Position: Right

In this section, we'll analyze the **blocking** algorithm. The analyzing took place in a well lightened room, with similar conditions of those in the library, to create maximum accuracy.

To evaluate the performance of the blocking algorithm, we computed the F1 Score and the Hit Rate. For this purpose, we calculated the True/False Positive/Negative rates. In out project, this is how they're defined:

True Positive (TP): The number of frames in which the algorithm detects blocking and there was one.

False Positive (FP): The number of frames in which the algorithm detects blocking, but there was none.

False Negative (FN): The number of frames in which the algorithm doesn't detect blocking, but there was one.

True Negative (TN): The number of frames in which the algorithm doesn't detect blocking, and there was none.

In order to compute the parameters, we started a frame counter, and we asked the player to block and unblock alternately. Then, we checked in an inner counter, in how many frames the program detected blocking. We let the program run for 273 frames. The number of frames in which the player blocked was 215. The program didn't detect blocking in any frame that the player was not blocking, and out of the 215 frames, it detected blocking 202 times.

That means that the results were:

$$TP = 202, \qquad FP = 0, \qquad FN = 215 - 202 = 13,$$
$$TN = 273 - 215 = 58$$

The Recall parameter measures the ratio of true positive predictions to the total number of actual positive instances in the dataset.

That means that $Recall = \frac{TP}{TP+FN} = \frac{202}{215} = 0.9395$

The Precision parameter measures the ratio of true positive predictions to the total number of positive predictions made by the model.

That means that $Precision = \frac{TP}{TP+FP} = \frac{202}{202} = 1$

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} = 2 * \frac{0.9395}{1.9395} = 0.96$$

$$Hit\ Rate = \frac{TP + TN}{TP + FP + FN + TN} = \frac{202 + 58}{202 + 0 + 13 + 58} = \frac{260}{273} = 0.95$$

However, when we lit the room directly on the sword, the algorithm's accuracy decreased. That is because, along with the algorithm explained above, we recognized the color of the sword, and the direct light on it caused the color, as presented through the camera, to change.

Let us compute the F1Score and the Hit Rate in that condition:

We checked 230 frames, while the player blocked through 165 of the frames. The results were:

$$TP = 68, \qquad FP = 0, \qquad FN = 165 - 68 = 97,$$
$$TN = 230 - 165 = 65$$

$$Recall = \frac{TP}{TP + FN} = \frac{68}{165} = 0.4121$$

$$Precision = \frac{TP}{TP + FP} = \frac{68}{68} = 1$$

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} = 2 * \frac{0.4121}{1.4121} = 0.58$$

$$Hit\ Rate = \frac{TP + TN}{TP + FP + FN + TN} = \frac{68 + 58}{68 + 0 + 97 + 65} = \frac{126}{230} = 0.54$$

When testing the effect of the "block" variable, which adds the dimension of continuity for the detection, as described in section 5 on the accuracy we found that when applying the new algorithm with the "block" variable, it performed well even with the new light condition.

When we raised the threshold of the Hough transform from 50 to 150, the accuracy, as expected, decreased. That is because the threshold is the minimum number of votes needed for a line to be detected, and the probability for it to detect the sword is now lower. The player

held the sword in the blocking position for 146 frames, but the algorithm succeeded in detecting that in only 106 frames. The same thing happened when we increased the MinLineLength. It makes sense because the line now should be longer for the algorithm to detect it.
When we decreased the ρ parameter, the accuracy also decreased. This is because now the algorithm is more sensitive to noise and lines may be cut so that one line could be detected as several lines. Along with the MinLineLength, it may result in less accuracy.

These are the results:

$$TP = 112, \qquad FP = 3, \qquad FN = 170 - 112 = 58,$$
$$TN = 206 - 112 - 3 = 91$$

$$Recall = \frac{TP}{TP + FN} = \frac{112}{170} = 0.6588$$

$$Precision = \frac{TP}{TP + FP} = \frac{112}{115} = 0.973$$

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} = 2 * \frac{0.641}{1.6318} = 0.78$$

$$Hit\ Rate = \frac{TP + TN}{TP + FP + FN + TN} = \frac{112 + 91}{112 + 3 + 58 + 91} = \frac{203}{264} = 0.76$$

Summary:

In conclusion, our project aimed to integrate gesture-based controls into the game "Infinity Blade". Using classical image processing techniques, we devised a system allowing players to wield a plastic sword as a controller, translating their physical movements into in-game actions in real-time. Our primary objectives were to ensure precision and reliability in converting human gestures into game inputs.

Throughout the project, we encountered various challenges and made several strategic decisions to overcome them. One of the decisions was to prioritize speed and efficiency in gesture recognition, leading us to work with image processing algorithms instead of machine learning approaches, due to their real-time performance. By employing techniques such as background subtraction with segmentation, HSV color masking, and the Kalman filter for object tracking, we achieved swift and accurate interpretation of player movements.

Specifically, our system tracked the player's sword movements using HSV color masking with a Kalman filter, enabling smooth and predictive tracking of the sword's trajectory. For blocking maneuvers, we implemented a Hough transform algorithm to detect a straight horizontal line above the player's head, signifying a blocking gesture. Additionally, dodging maneuvers were recognized by analyzing the movement of contours after background subtraction, with left and right dodges inferred based on predefined thresholds.

Despite encountering challenges such as sensitivity to lighting conditions, we knew how to overcome. One of the solutions was introducing adaptive mechanisms like the "block" variable to maintain robust performance across diverse environments.

In summary, by leveraging image processing techniques and strategic algorithmic design, we successfully translated physical gestures into meaningful in-game actions, enhancing the overall gameplay dynamics of "Infinity Blade."

9. Bibliography:

- https://opencv.org/
- https://archive.org/details/infinity-blade-pc
- Suzuki, S. and Abe, K., Topological Structural Analysis of Digitized Binary Images by Border Following. CVGIP 30 1, pp 32-46 (1985)
- Zoran Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 2, pages 28–31. IEEE, 2004.
- G. Welch and G. Bishop, "An introduction to the Kalman filter, Tech. Rep., 1995.