Using Computer Vision to Play Super Hexagon

Travis Geis
Stanford University
tgeis@stanford.edu

Schuyler Smith
Stanford University
skysmith@stanford.edu

Rohit Talreja
Stanford University
rtalreja@stanford.edu

Abstract

We present work towards an end-to-end system for playing the game Super Hexagon[1] on a mobile phone. Unlike previous methods, our approach captures real-time video from a webcam observing the device screen, rather than from low-level operating system display buffers. Because it uses only visual observations of the game device, our system does not assume anything about the device itself such as its operating system, screen size, or resolution.

The video processing pipeline is split into two stages: First, it isolates the phone screen from video using a combination of thresholding, RANSAC and perspective transformations. Second, it extracts game features from a single frame using adaptive thresholding, convolutional blob detection, and ray casting.

This process is reliably able to locate the phone on a uniform white background. To correct for perspective distortion of the screen image, we assume the phone is in landscape orientation relative to the camera. Additionally, we are able to locate the player character and the closest obstacles very reliably. Although the game runs at 60 frames per second, our maximum processing speed is around 20 frames per second, with a latency of 2-3 frames. The latency of the system comes primarily from video encoding in the webcam and decoding on the computer. Using the extracted game features, we use a rule-based AI to move the player in the optimal direction to avoid the nearest obstacle.

1. Introduction

Game-playing agents are of interest to artificial intelligence researchers since researchers are often able to repurpose the techniques pioneered for solving open-ended game dilemmas for solving other traditionally-hard problems. While most research in game-playing agents centers on the artificial intelligence tasks and uses game APIs to manage the game state directly, in our project we are investigating a system where the game-playing agent receives only the visual output of the game, as would a human player. The techniques involved in playing the game as a human would could allow the system to play other games in the future, even if they only exist as mobile apps.

We focus specifically on Super Hexagon, a fast-paced mobile and PC game where the player must pilot a small character through the scene to avoid geometric obstacles (Figure 1). We chose this game because it has relatively simple and repetitive 2D features, only one degree of freedom (circular rotation around the hexagon in center of the screen) and the only objective is staying alive by avoiding obstacles. As a result, designing an AI to play the game is relatively simple (compared to other video games), but the games fast-paced nature still makes it an interesting computer vision challenge. All objects on screen except the player character rotate around center with the same angular speed, though the rotation changes randomly as the game progresses.

We present both a video processing pipeline to isolate the phone screen from webcam video and extract the relevant features as well as a decision-making AI to play the game based only on these extracted features. For testing purposes, we use both pre-captured gameplay videos and live video feeds of a phone screen. We enable the AI to play the game directly on the phone by simulating keypresses over a bluetooth connection.

2. Background

2.1. Previous Work

There are several challenges associated with real-time video feature extraction for game-playing: how to recognize an object of a particular size and shape in a possibly-crowded video frame, how to determine the orientation of that object in order to correct for perspective transformation, and finally how to do so quickly enough to keep pace with gameplay.

Due to the visual simplicity of Super Hexagon, a relatively simple thresholding process allows us to separate the foreground features from the background. Current state-

[1] Super Hexagon. Terry Cavanaugh. 2012. Video Game.

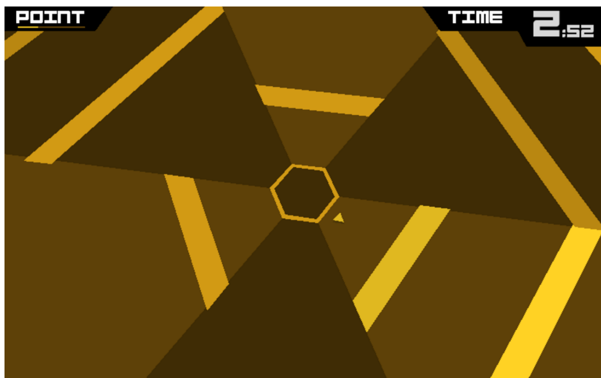

Figure 1. Super Hexagon. The player, the small triangle near the center of the screen, must avoid the impending walls.

of-the-art algorithms use convolutional neural networks (CNNs) for object recognition; we had originally considered this approach as well but decided against it due to lack of annotated training data. Additionally, since we only look for one object in the frame, a phone, it did not make sense to use powerful CNNs that are built to differentiate between hundreds or thousands of object classes. However, there may be ways to reuse some of the features that make CNNs fast at processing data once they have been trained.

One phone-isolation procedure we investigated makes use of OpenCVs contour-finding function for binary images, which is based on the algorithm proposed by Suzuki and Abe[4]. This is an iterative pixel-labeling algorithm that attempts to connect groups of 1-pixels that surround, or are surrounded by, groups of 0-pixels. The algorithm performs a raster scan of the image and locates possible pixels for a border, which is as simple as detecting whether the pixel has value 1 while a neighboring pixel has value 0. Next, the pixel is given an integer label to keep track of which border it belongs to. With each subsequent starting pixel a decision is made whether to add it to an existing border or create a new border with a new label. The algorithm also checks when border segments connect. If its determined that two segments are part of the same border, pixel labels are reassigned to make them one border.

The second phone-isolation method we tried relies on the background subtraction algorithm proposed by KaewTraKulPong and Bowden[2]. Assuming that the phone is stationary, the foreground of the image (the phone screen in our project) will change more often than the background. Hence, we can use the changes between frames to update and threshold a reference image. The result of the algorithm is a binary segmentation of the image that highlights moving objects. The largest downside of this approach is that it cannot handle changes in illumination in the scene.

We also found two existing projects that attempted to play Super Hexagon[5][3]. However, both focus more on the game AI than the video processing steps. In particular, both projects rely on hacking the Super Hexagon source code and OS data structures to gather game state. As a result, they are less portable and less generally applicable than our approach.

As for the specific video processing steps, the more complete project, led by Valentin Trimaille, binarizes the image to remove background and detects obstacles by casting rays from the center of the screen until they hit an obstacle or edge of screen. This is similar to our approach. Their approach to finding the player character relies on its triangular shape; they detect all contours and then assume the only contour with 3 points is the player. This approach does not work well for us because noise in the binarization/thresholding process can create false positives. Our approach, described in section 3.2.2, is much more robust.

2.2. Our Contributions

As described in the previous section, the prior work in Super Hexagon AIs was based on acquiring the game video from low-level display drivers, an approach that does not generalize well between different devices and operating systems. This approach also does not work well for mobile or console games, as there is no API to extract single frames from their graphics processors. Mobile games already hold a significant market share and continue to gain in popularity, so it is imperative to develop an alternate video acquisition and processing method, which is the purpose of our project.

We have attempted to create a generic video-processing workflow that can be used to extract relevant features from games played on a mobile phone over a webcam video stream. Although we are developing this process specifically for Super Hexagon, we believe that a majority of the feature extraction components can be generalized to play other 2D games.

3. Solution

3.1. Summary

Our game-playing process (Figure 2) consists of three major steps: 1) isolating the phone screen from webcam video feed, rectifying the phone screen to correct for perspective transformation and removing the background (red border), 2) extracting the player character and obstacles from each frame (blue border), and 3) choosing the best direction to move based on the location of the player in relation to the obstacles (green border). These steps correspond to the domains of computer vision, image processing, and AI respectively.

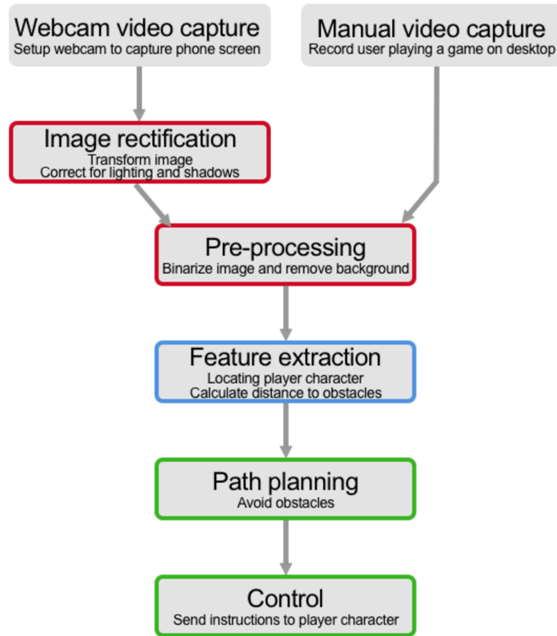

Figure 2. Pipeline for playing Super Hexagon.

3.2. Isolating the Phone and Correcting for Perspective

Our original goal was to devise a procedure that can locate the phone screen in any orientation and position within the frame, even with other objects in the field of view. However, if the phone orientation is completely unknown, we encounter perspective ambiguity in trying to recover its orientation. To simplify the problem, we assume that the phone is in landscape orientation relative to the camera, and that it is stationary. However, to keep the system somewhat flexible, we do not assume that the plane of the phone screen is aligned with the image plane of the camera, and we allow small rotations of the phone.

We implemented two different systems for identifying the boundaries of the phone screen in a video stream: first, a simple thresholding scheme; and second, a system using background subtraction to locate the animations on the phone screen. After finding the contours of the screen, we use RANSAC [1] to find the corners of the best-fit quadrilateral.

To overcome the difficulties of truly generic phone-screen detection, we make two simplifying assumptions for the thresholding approach. First, we assume that the phone is the only object in view of the camera and it is placed on a uniform light background, such that thresholding is an effective means of isolating the silhouette of the phone. Second, we assume that the phone is in landscape orientation, though it need not be perfectly axis-aligned. The thresholding algorithm proceeds as follows:

1. Convert the image to grayscale
2. Binarize using an inverted threshold function, which sets

$$\text{new pixel value} = \begin{cases} 255 & \text{if pixel value} < 40 \\ 0 & \text{otherwise} \end{cases}$$

   This has the effect of discarding the light-colored background and locating the silhouette of the phone.

3. Find all closed contours in the frame using OpenCVs findContours function, which uses [4]. Since the phone is the only object in the frame, the resulting contour gives its silhouette.

The thresholding approach to locating the phone is simple and fast, though it stipulates that the phone is on a light, uniform background. To remove the background assumption, we can instead assume that the phone is stationary, and that its screen will display vibrant animations during gameplay. Because the contents of the screen are in motion, we locate the screen using a background subtractor. Considered simply, the background subtractor maintains a reference image over time, then subtracts each new frame from this reference image and thresholds the result to locate the foreground region. We use OpenCVs BackgroundSubtractorMOG, which implements the mixture-of-Gaussians background subtractor as proposed by [2].

The above methods to isolate the phone screen yield the set of points along the contour. To remove the background of the frame and align the phone screens contents to the axes of the image, we apply a RANSAC-based approach to recover the four corners of the screen. Using these corners, we can warp the screen contents to fill the frame. The algorithm follows:

1. Using RANSAC, find four points along the perimeter of the contour that enclose an area closest to the area of the original contour. The RANSAC procedure tracks the four best points over 1000 iterations. In each iteration,

   (a) Choose four points at random from the contour
   (b) Calculate the area enclosed by the 4 points
   (c) If the area is greater than the current best set of points, replace the best set of points

2. Maintain a ring-buffer of the recovered quadrilateral, and take the average of the buffer to be the current screen quadrilateral, a projected rectangle.

3. Given the four corners of the screen and the known positions of the image frames corners, we find the perspective transformation to align the contents of the phone screen to the image frame. Images: isolated.png

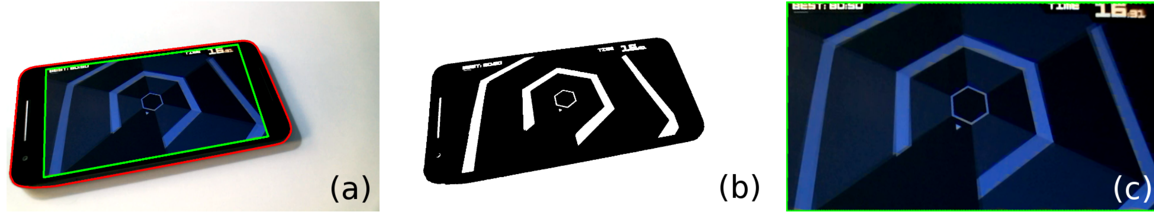

Figure 3. Process of phone screen isolation. (a) is the original image, with phone contour shown in red, and ground truth screen quadrilateral shown in green. (b) is the thresholded image. (c) Is the recovered screen image.

3.3. Extracting Features

Once the phone screen is isolated and corrected for any perspective transformation, we normalize its resolution and extract the relevant game features. First, we use an adaptive threshold to binarize the image and isolate the foreground – the player characters and obstacles – from the background. The webcam performs automatic exposure adjustments as the screen content changes, though it usually lags significantly, so our method must be effective in a very wide range of lighting situations. Starting with a grayscale image, it calculates the median brightness value and uses that as a noise-resistant estimate for the background color. Then, the threshold was set empirically to a brightness 1.5x this median value.

At this point, only the player character and obstacles remain, and the next step is to determine their exact location. We first detect the player character, which conveniently is always about the same size and stays near the center of the game. We created a blob detector by convolving a 30x30 pixel kernel with the center 120x120 pixels in the image. The kernel has positive weight in the central region corresponding to the expected size of the player, and negative weight surrounding. This way regions larger than the player will receive negative weights after convolution, and the pixel with highest weight will be at the center of the player, providing excellent localization. Results can be seen in Figure 11. After locating the player, the player is masked out to ensure it doesnt interfere with obstacle detection.

In an effort to make the processing pipeline as quick as possible, we only consider the closest obstacles in each direction from the player character with the assumption that the AI will react quickly enough that lookahead isnt necessary. Unfortunately in practice we found that this wasnt achievable. We project rays from the center of the screen in 10 degree increments and keep track of the length of each ray at the time it meets the first obstacle. In the early levels, its possible that a ray reaches the edge of the screen before hitting an obstacle, in which case we consider the of the screen to be an obstacle even though it doesnt move toward the player. In testing we found that this was preferable to capping the measured distances (such as by restricting the rays to a circle inscribed in the phone screen, which thus discards potentially useful information from the sides of the screen). The angle and length of each ray is used by the AI to determine the proper action for the frame.

3.4. Game AI

Developing a robust game AI was not the focus of our project. Our goals instead were to develop a simple AI that would be easy to reason about, and adequately showcase the rest of our pipeline.

The game AI is stateless and can choose one of three options (move counterclockwise, move clockwise, or stay still) in each frame. Because we did not introduce any lookahead behavior, the AI does poorly with obstacles that require advanced planning. This also leads to it being extremely susceptible to latency, as reaction time is the biggest factor in performance. We use pre-measured values for player rotation speed and obstacle movement speed to determine how far the player can move in either direction without dying. Then, we move in the direction with the deepest reachable opening. That is, the AI chooses to move toward the angle with the furthest distance to an obstacle, restricting our search to only angles that are feasible.

This was an attempt to mimic the goal-oriented play of human players, but the AI is significantly hamstrung by latency and its shallow lookahead.

4. Experiments

We tested our video processing pipeline using both live video feeds for qualitative results (Figure 4) and pre-captured video for repeatable quantitative tests of the different methods. To assess the AIs performance, we used live video to see how long the AI could play independently and pre-captured video of a human player to assess the differences between AI and human performance.

4.1. Isolating the Phone

To test our phone-isolation algorithm we compared its output to hand-annotated ground truth locations of the corners of the screen in four test videos (Figure 5). We used the Jaccard Index to determine the amount of similarity be-

Figure 4. Experimental setup. The phone, right, is filmed by the webcam, which is connected to the laptop. In this test, the laptop is drawing the extracted bounding box on the raw webcam footage.

Figure 5. Screenshots from the four videos used to test bounding box extraction. Referred to by the numbers 1-4, clockwise from top right, in the tables below.

tween the hypothesized and actual screen locations. A ratio of one would indicate that the algorithm had perfectly located the screen, and less than one would indicate an imperfect match.

After recovering the screen quadrilateral, we use three metrics to evaluate the performance of the two phone-screen isolation methods: the Jaccard index of the recovered quadrilateral, the average distance of the corners rectangles from those of the ground truth, and the average corner velocity, which we define to be the average distance the recovered quadrilateral's corners move between each frame of the video. The Jaccard index, defined as

$$J(H,G) = \frac{\mid H \cap G \mid}{\mid H \cup G \mid}$$

where H is the hypothesized quadrilateral, and G is the ground truth quadrilateral. gives an idea of the correctness and completeness of the isolated screen area, and a value close to 1 is ideal. The corner distances should be low for correct screen recovery, and can indicate screen rotations and skew. Because our AI needs stable video frames, we also strive for low corner velocity.

Using ring-buffer sizes of 1, 35, and 50 frames, we present the above metrics in Figures 6 and 7. Both screen-isolation methods offer relatively high Jaccard Indices, and low corner distances for all video files except file 1. In video file 1, the phone is sufficiently rotated away from landscape orientation that the recovered screen-bounding quadrilateral has ambiguous vertex order; the vertex-ordering procedure cannot accurately determine which corners correspond to the four ground-truth corners. The high corner distances for file 1 reflect the incorrect vertex correspondences, illustrating the importance of our assumption that the phone is in landscape orientation.

Despite the similar Jaccard and corner-distance scores on the other files, we conclude that the thresholding method produces more reliable estimates of the phone screen for the purposes of the AI agent. The background-subtraction method suffers from higher corner velocity, due to its inability to select the corners of the game screen, which contain stationary UI elements. This higher corner velocity corresponds to much more jitter in the recovered image, which suggests motion on the phone screen even where there is none.

	File	Jaccard Index	Corner Dist	Corner Vel
Buffer 1	1	0.743 ± 0.015	516.8 ± 52.2	24.12
	2	0.806 ± 0.020	62.2 ± 9.3	18.89
	3	0.771 ± 0.015	58.8 ± 7.2	16.56
	4	0.785 ± 0.013	77.7 ± 8.1	14.69
Buffer 35	1	0.764 ± 0.017	515.9 ± 10.2	0.98
	2	0.820 ± 0.010	58.5 ± 2.9	0.78
	3	0.782 ± 0.003	55.6 ± 1.6	0.69
	4	0.792 ± 0.002	77.1 ± 1.3	0.74
Buffer 50	1	0.772 ± 0.018	510.6 ± 11.1	0.90
	2	0.820 ± 0.010	58.8 ± 3.0	0.62
	3	0.782 ± 0.002	55.5 ± 1.1	0.55
	4	0.792 ± 0.002	77.0 ± 1.3	0.62

Figure 8. The numerical scores used to generate the graphs in Figure 6, with standard deviations.

4.2. Extracting Features

Feature extraction proved to be very reliable. It was evaluated on a 50 second webcam test video (a recording of a human playing the game, using our normal experimental setup) by comparing the algorithms results to hand-annotated player positions. Results can be seen in Figure 10

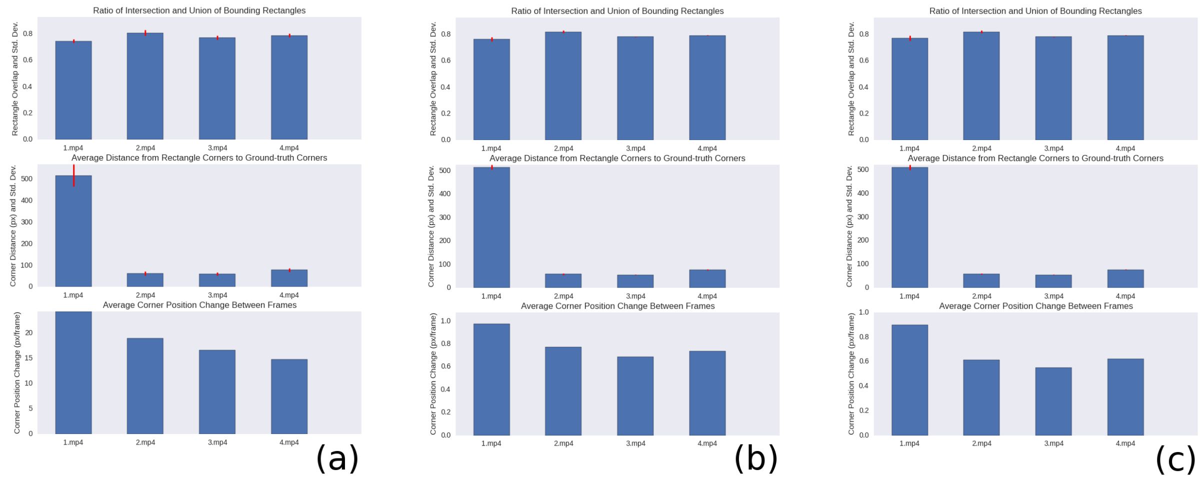

Figure 6. Top to bottom: Jaccard Index, corner distance, and corner velocity for the thresholding method in each video file and (a) a buffer size of 1 frame, (b) a buffer size of 35 frames, and (c) a buffer size of 50 frames.

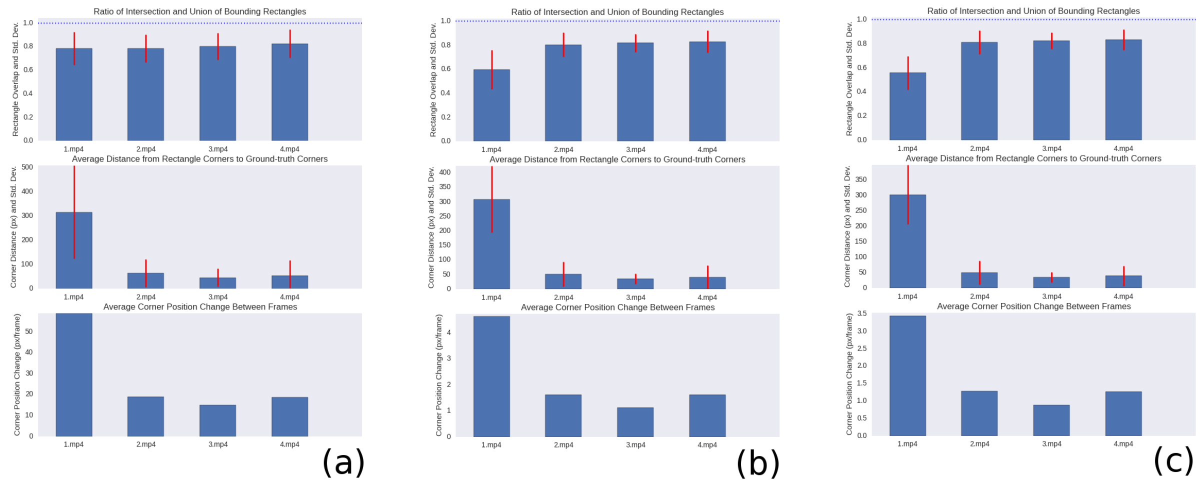

Figure 7. Top to bottom: Jaccard Index, corner distance, and corner velocity for the background subtraction method in each video file and (a) a buffer size of 1 frame, (b) a buffer size of 35 frames, and (c) a buffer size of 50 frames.

The player detector reports no detection when no pixels pass a threshold. This was the most common failure mode for the algorithm, and only occurred when the player was touching a wall and was thus difficult to isolate. True false detections were much less common, and usually seemed to be the result of camera noise or the camera refocusing.

Measuring the distances to obstacles was even more reliable, since the obstacles are larger and slower moving than the player character. We didnt notice any mistakes in our test videos. This step initially comprised a significant fraction of our processing time, but we found that reducing the number of rays cast and increasing the stride to 5 pixels made the cost negligible and didnt significantly impact the performance of the AI.

4.3. Game AI

Finally, we evaluated the entire system by measuring its ability to play the game in real time. Our results were significantly better than a random player, but worse than wed hoped. We also compare the system to a human player in Figure 12. For this test the human player also played through the webcam, to mimic the latency faced by the AI and make the comparison as fair as possible (the same human can consistently score greater than 60s on the original

	File	Jaccard Index	Corner Dist	Corner Vel
Buffer 1	1	0.785 ± 0.139	315.1 ± 192.1	58.76
	2	0.783 ± 0.120	63.9 ± 56.7	18.90
	3	0.802 ± 0.115	45.1 ± 37.4	14.91
	4	0.824 ± 0.120	52.1 ± 65.2	18.58
Buffer 35	1	0.599 ± 0.162	308.4 ± 114.4	4.63
	2	0.805 ± 0.100	51.6 ± 42.1	1.61
	3	0.819 ± 0.074	35.4 ± 18.3	1.12
	4	0.831 ± 0.093	41.3 ± 40.2	1.62
Buffer 50	1	0.558 ± 0.138	302.5 ± 96.1	3.43
	2	0.810 ± 0.098	49.5 ± 38.6	1.28
	3	0.824 ± 0.067	34.3 ± 16.5	0.88
	4	0.833 ± 0.085	39.7 ± 32.6	1.27

Figure 9. The numerical scores used to generate the graphs in Figure 7, with standard deviations.

	Frames	Percent
Correctly identified	1302	93.7%
Incorrectly identified	24	1.7%
No match	64	4.6%

Figure 10. Player character localization accuracy.

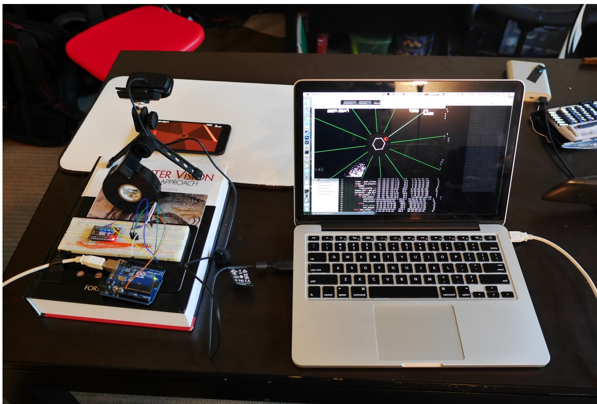

Figure 11. Experimental setup for the integration tests. The arduino and bluetooth module at bottom left were used to transmit controls to the phone.

	No Movement	Random Movement	This Paper	Human
Mean (30 trials)	3.52 ± 0.19	3.35 ± 0.20	4.29 ± 0.27	9.09 ± 0.53

Figure 12. Player character localization accuracy.

game, so this is a significant handicap).

The AI tended to fail in predictable ways. Super Hexagon often generates walls with small openings, and the system almost never had enough time to escape these if it didnt start near the opening. The latency of the game also led the player to overshoot its openings very often: in the 2-3 frames between when the system decides to stop moving and when the game actually reacts, the player character has usually passed the target location.

5. Conclusions

In this paper we sought to play Super Hexagon from a webcam video feed. We attempted to make the first stage of our pipeline, phone-localization and perspective correction, as generic as possible. However, we had to make several simplifying assumptions about background content and phone orientation. With these simplifying assumptions we were able to correctly and reliably identify the phone screen.

In the second stage of our pipeline, we processed the transformed phone screen to extract relevant game features. This stage showed strong robustness to noise in the image, and proved more reliable than the first stage.

Finally, we developed a simple AI to use our extracted features to play the game. Developing a robust game AI was not the focus of our project. Our goals instead were to develop a simple AI that would be easy to reason about, and adequately showcase the rest of our pipeline. It achieved results significantly better than random chance, but much worse than our human baseline.

Given additional time, we would reduce the pipeline latency, add state to the AI game-playing agent, and develop a more robust and efficient method for localizing the phone screen.

Our final processing pipeline has throughput of around 25 FPS, which gives a per-frame processing time of 40 ms. However, the sum of webcam processing time, transfer latency and video decode time on the computer add up to greater than 100ms. As a result, our AI is receiving information that is around ten game-frames out-of-date. Since latency is the biggest detriment to AI performance, reducing the latency could vastly improve the game-playing results. Using camera systems designed for real-time applications such as robotics or surveillance could be a convenient avenue for reducing latency.

For simplicity, we designed the game AI to be stateless and only take the closest obstacle into account when choosing whether to move and which direction to move. This is inefficient in most situations, because moving in the suboptimal direction may actually put the player in a better location to deal with the next layer of obstacles. We would have liked to enhance the AI to use more sophisticated path-planning algorithms, such as Dynamic-A* that update immediately based on the appearance of new obstacles [Likhachev]. A predictive AI may also help mitigate

some of the processing latency by extracting more information from each frame such that it plans motion for several frames ahead rather than for the current frame.

Lastly, we would like to devise a more efficient and deterministic method of locating the phone screen in the video stream. Our algorithm uses a bounding contour [Suzuki] to locate the phone, followed by RANSAC to turn that into the best bounding rectangle. RANSAC causes some noise in the algorithm, which we compensate for by averaging the bounding rectangle over several frames.

While our AI agent did not perform as we had expected, we were able to use computer-vision techniques to begin a system for playing unmodified games on mobile devices. With reduced latency and more robust video capture techniques, we hope that our processing framework could serve as a platform for future advances in game-playing agents.

The code for our project is located at: https://bitbucket.org/ottobonn/super-haxxagon/

References

[1] M. A. Fischler and R. C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[2] P. KaewTraKulPong and R. Bowden. An improved adaptive background mixture model for real-time tracking with shadow detection. In *Video-based surveillance systems*, pages 135–144. Springer, 2002.

[3] C. Pineur. Super hexagon bot, 2015.

[4] S. Suzuki et al. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985.

[5] V. Tremaille. Super hexagon bot, 2015.