Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Line Rider in Augmented Reality

CSU44054 Augmented Reality Report

Eoin Gogarty

Student No. 14320042

May 31, 2021

Contents

List of Figures

Chapter 1

Introduction

1.1 Project Overview

This project implements an augmented reality (AR) version of the popular online game Line Rider [1] which is shown in Figure 1.1. In the normal game the player draws lines on a blank screen which the character rides upon, usually on a sledge, until they crash. The character can then be made to do somersaults and other fun acrobatics purely based on the shape of the drawn track. The lines can be removed, edited, or added to with the mouse cursor and the character is then reset to the top of the track on every new run.

In this project the augmented reality version is overlaid on a piece of paper, or any flat surface that can be drawn on with a pen, pencil, or marker. This drawing surface is tracked using an ARTag and the camera of a phone or other device. The game is viewed by the player through the screen of the device, while they draw and erase in real time on the surface with a pen, pencil, or marker. The steps involved in playing the game are as follows: print out the ARTag and place it on the surface, go to the Line Rider AR web page on the device, draw the track, and finally point the camera at the track and click play. The character will appear on the page and ride along the lines.

No previous implementations of this idea were found online so this seems to be an original concept. This was a large motivating factor for this project; to complete and publish the game before a third party produced their own version. The project successfully resulted in a working prototype game, with all the specified functionality.

1.2 Approach

The AR game is written in JavaScript and is playable on any modern device with web browser capabilities and a camera, including phones and tablets. Once the web page for the game is loaded, no more interaction with the internet or a server is needed, meaning

[1] Line Rider (www.linerider.com) is a classic sandbox game where you draw a track for the sledder to ride on

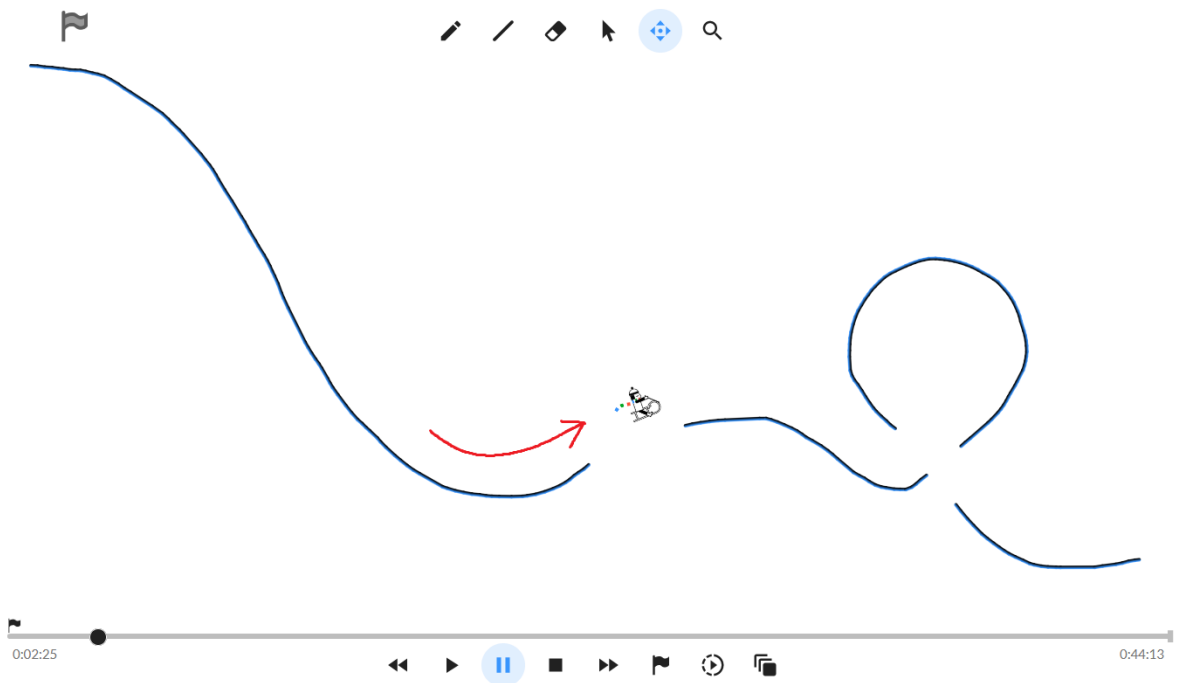

Figure 1.1: *A simple track from the official online Line Rider game, drawn with the cursor of a mouse on a screen.*

that the game can in essence be played offline. The JavaScript libraries used in this project for 3D graphics and marker tracking are Three.js [1] and THREE AR [2], the latter using ARToolKit as a back end. OpenCV.js [3] is used for the image processing. The main steps involved in the implementation are described in detail in Section 3 and are summarised as follows:

- drawing surface detection
- drawing surface extraction
- line and track extraction
- game engine physics
- player AR visualisation.

1.3 Motivation and relevance

This project combines many different areas of augmented reality, 3D vision, image processing, and browser programming which makes it both challenging and interesting. Building a working pipeline with all these different technologies is a greater task than the sum of its parts. The outcome of this project is a portable, universal, and entertaining game which can be played on almost any device. Importantly, it does not require an

expensive VR headset to play which lowers the barrier to entry for people who only have a smartphone.

The motivation for this project is to merge the physical interaction with the world, namely drawing on a piece of paper, with cutting-edge augmented reality, to bring a new type of game to the public. Line Rider is a famous game which has existed for years on the internet and has garnered millions of players, however this is the first time that it has been adapted to augmented reality and the first time that a game of its type has been brought into the physical world.

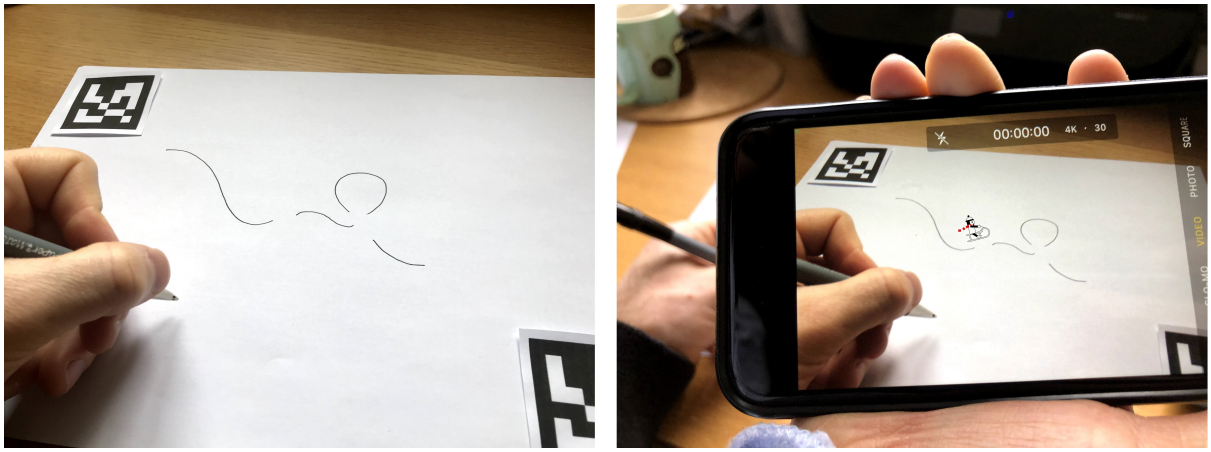

Figure 1.2: *Concept images of what the final Line Rider AR game could look like and how it could function.*

Chapter 2

Theoretical Background

2.1 Marker detection

THREE AR uses ARToolKit to perform marker detection. It scans the video stream and searches for quadrilaterals, representing a potentially skewed square. It converts these quadrilaterals to squares using methods similar to that described in Section 2.3. It then compares these squares to the chosen ARTag to find a match. Once the matching pattern is found, the position and orientation of this marker in 3D space are calculated relative to the camera. For this project, and for many marker-based augmented reality projects, the marker is set to be the origin of the scene. All other objects are positioned relative to the marker, allowing them to have spatial awareness in 3D space.

ARToolKit uses history functions over sequences of previous video frames to improve the tracking stability of the marker patterns. It decreases the accuracy of the marker detection but improves the smoothness of the 3D movement. This is useful for displaying and animating the character in this project, but is not used when scanning the track.

2.2 Coordinate space conversion

The image processing and character animation in this project both require the conversion of points in the 3D space of the scene to be projected onto a 2D space, from the point of view of the camera. Relevant points in the scene are stored in the local coordinate system of the ARTag. These first need to be converted to the real world 3D space, using the marker's transformation matrix. Once in this real world space, the points are cast onto the camera screen using the camera's transformation matrix and projection matrix. Some scaling is then required to match the dimensions of the video stream.

2.3 Perspective transform

The implementation of the track scanning allows the camera to capture the drawing surface from any angle. This means that the surface needs to be transformed into a

rectangle before image processing is applied as in Figure 2.1. A perspective transform is used to do this. The on-screen positions of the four corners of the page are needed along with the coordinates of a perfect rectangle, with the same width-to-length ratio as the actual flat drawing surface. A transformation matrix can then be calculated using the two sets of quadrilaterals corners. This is done by solving the four pairs of following equations, a pair for each corner, where (X, Y) and (U, V) are the source quadrilateral corners and target quadrilateral corners respectively:

$$U = \frac{a_0 + a_1 X + a_2 Y}{1 + c_1 X + c_2 Y}$$

$$V = \frac{b_0 + b_1 X + b_2 Y}{1 + c_1 X + c_2 Y}$$

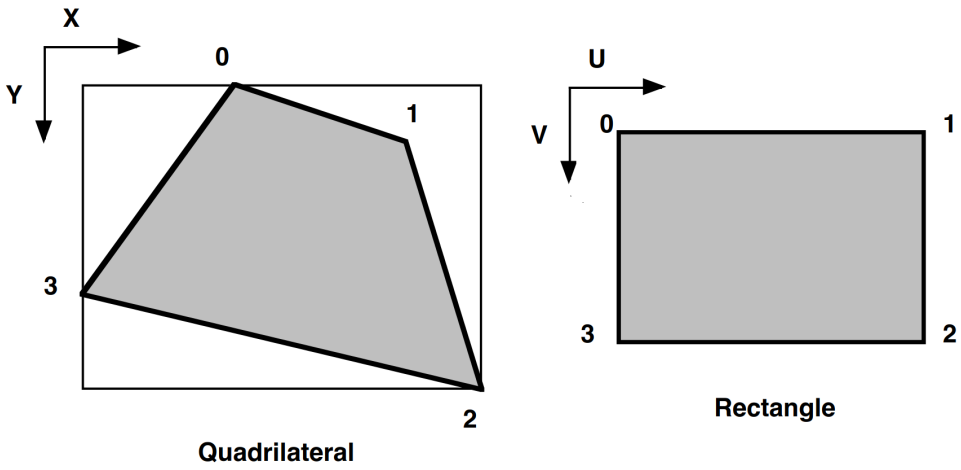

Figure 2.1: *A visual example of the effect of perspective transform in warping a quadrilateral into a perfect rectangle.*

Chapter 3

Implementation

The template shown in Figure 3.1 is used for playing the game. Once printed out it can be drawn on, within the bounded area, and used with the AR device. The drawing area width is three times the marker width, while the drawing area length is five times the marker width. This bounding box is mainly used for testing purposes and for constraining and highlighting the drawing area, but it would be possible to use the fiducial marker by itself on any blank surface. The Hiro marker pattern, also shown in Figure 3.1, is used for tracking the surface in this project but any other ARTag could be used.

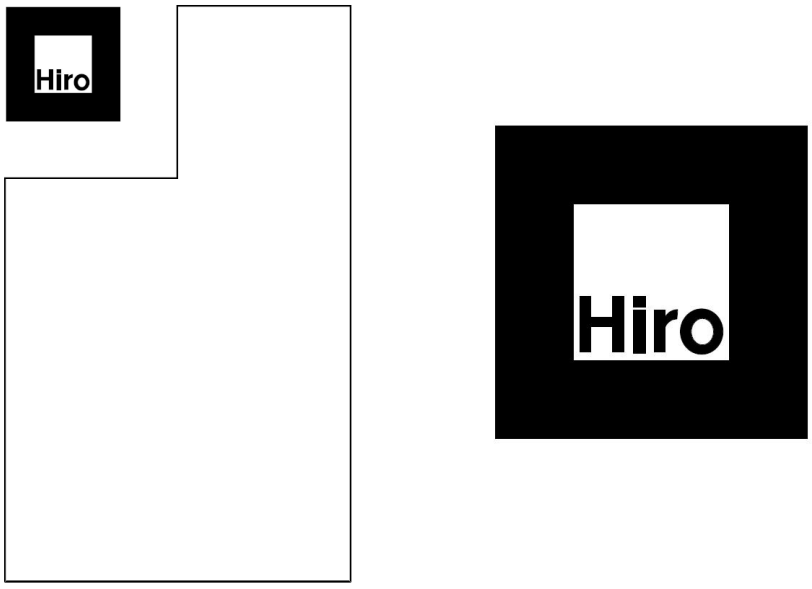

Figure 3.1: *Left: Page template with Hiro marker. Right: High-resolution Hiro marker.*

3.1 Drawing surface detection

After the player has drawn their track and wishes to play the game, they point their phone at the drawing surface. The device will then detect the location of this surface in

3D. The Three.js library allows a connection to the webcam on the device and forwards the video stream to the web browser screen. The THREE AR library has pattern-tracking functionality and can locate the position and rotation of the Hiro marker and therefore the top-left of the page. The page corner locations are known relative to the marker, based off the constant width and length of the drawing area template, but still need to be converted to real world coordinates.

3.2 Drawing surface extraction

With the surface and its corners located, the local coordinates of the corners in the marker space need to be converted to real world coordinates and then to 2D video frame coordinates. This conversion is carried out with Three.js using the transformation and projection matrices of the marker and the camera, as described in Section 2.2. The current video frame is captured and the corners of the page can be used to calculate the perspective transformation matrix, mentioned in Section 2.3. This matrix is then used by OpenCV.js to warp the skewed picture of the page into a perfect rectangle for further image processing. This is necessary as the camera may view the drawing area from any angle. This process is shown in the graphics below in Figure 3.2.

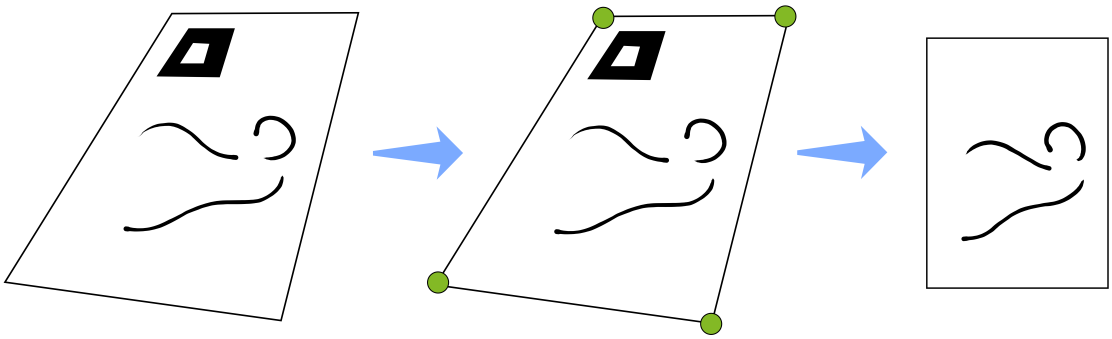

Figure 3.2: *Diagram of desired page extraction process, from corner detection to perspective transform.*

3.3 Line and track extraction

This part of the game pipeline is focused on using OpenCV.js to perform image processing on the drawing area to extract an accurate vectorised representation of the track. Many combinations and permutations of image processing algorithms and parameters were tried and tested as listed in Section 4.3 to achieve the optimum results. The final sequence of algorithms and transformations are as follows:

- grayscale conversion
- Canny edge detection → this roughly extracts all the lines in the frame, but can detect two bordering lines if the pen used is quick thick
- masking of Hiro marker and border lines → this is performed after Canny edge detection once the image is in a binary representation so that these areas can just be set to black
- morphological transformation - image dilation → this thickening of the lines seeks to merge these incorrect double lines from the Canny edge detection, and to smooth over any missing segments in a continuous line
- morphological transformation - image erosion → this counteracts the previous dilation step so that the detected contours in the next step are as close to the original line thickness as possible
- contour identification → this converts the processed binary lines into vectorised interconnected points
- contour polygon approximation using the Douglas-Peucker algorithm [4] → the detected contours contain too much granularity and detail, and are smoothed by this step into longer line segments.

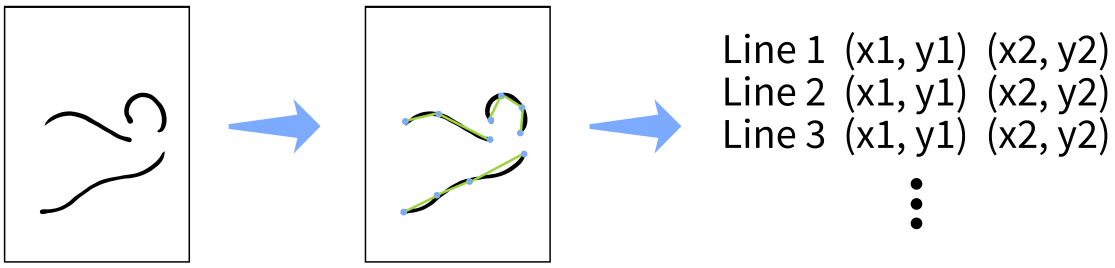

Figure 3.3: *Diagram of desired track and line extraction process, from RGB image to an array of line vectors.*

3.4 Game engine physics

The result of the final polygon approximation step is an array of points representing the line segments. These are then fed into the Line Rider physics engine, taken from a previous version of the official online game [5]. This game engine library was compiled

from a Node.js implementation into a static client-side script, using Browserify [1], to ensure that no server or continuous internet connection would be needed when playing the game. The game engine performs a full simulation of the track and returns an array of all the player positions for each time step.

3.5 Player AR visualisation

The player coordinates from the physics engine are converted from the video frame dimensions back into the drawing area dimensions. This provides a pair of x-y coordinates over time which are used as the positional arguments for a 3D character object. No other transformations are required to map from the 2D space as the the drawing surface is assumed to be flat and all coordinates are relative to the marker anyway. This character object is created using Three.js and can be animated with this array of coordinates, using the popular Anime.js [6] JavaScript animation library and its keyframe functionality.

[1] Browserify (browserify.org) is an open-source JavaScript tool that allows developers to write Node.js-style modules that compile for use in the browser

Chapter 4

Experiments and Results

4.1 JavaScript libraries

4.1.1 Perspective transform and image processing

Several JavaScript libraries were tested for performing the perspective transform, required for the page extraction stage. Initially glfx.js [7] was used, but was replaced in favour of JSFeat [8] because of the additional image processing methods included in the code package. However, JSFeat did not include straightforward methods for image reading and writing, nor individual pixel manipulation. It also lacked an implementation of the contour function needed for the line vectorisation. For these reasons it was decided to use OpenCV instead, which required a re-write of the perspective transform and image processing code to factor out the previous libraries. The results of the OpenCV perspective transform is shown below in Figure 4.1.

4.1.2 3D rendering and marker detection

In the initial development stages of the project, the JavsScript libraries used for 3D object rendering and 3D object detection were A-Frame [9] and AR.js [10] respectively. AR.js uses ARToolKit as the back end to perform marker tracking. When the image processing library was changed to OpenCV.js, for reasons listed in Section 4.1.1, this caused compatibility issues between the 3D libraries and OpenCV. To fix this, THREE AR and Three.js were included in the project to replace AR.js and A-Frame. This required a large re-write of the code for the augmented reality portion of the project.

4.2 Coordinate conversion issues

Experiments with the conversion of 3D to 2D coordinates brought up several problems linked to incorrect scaling with the device used. The nature of the universal browser approach to this project means that various types of devices can be used to play the game, all with different camera and screen dimensions. Users could also resize the windows on

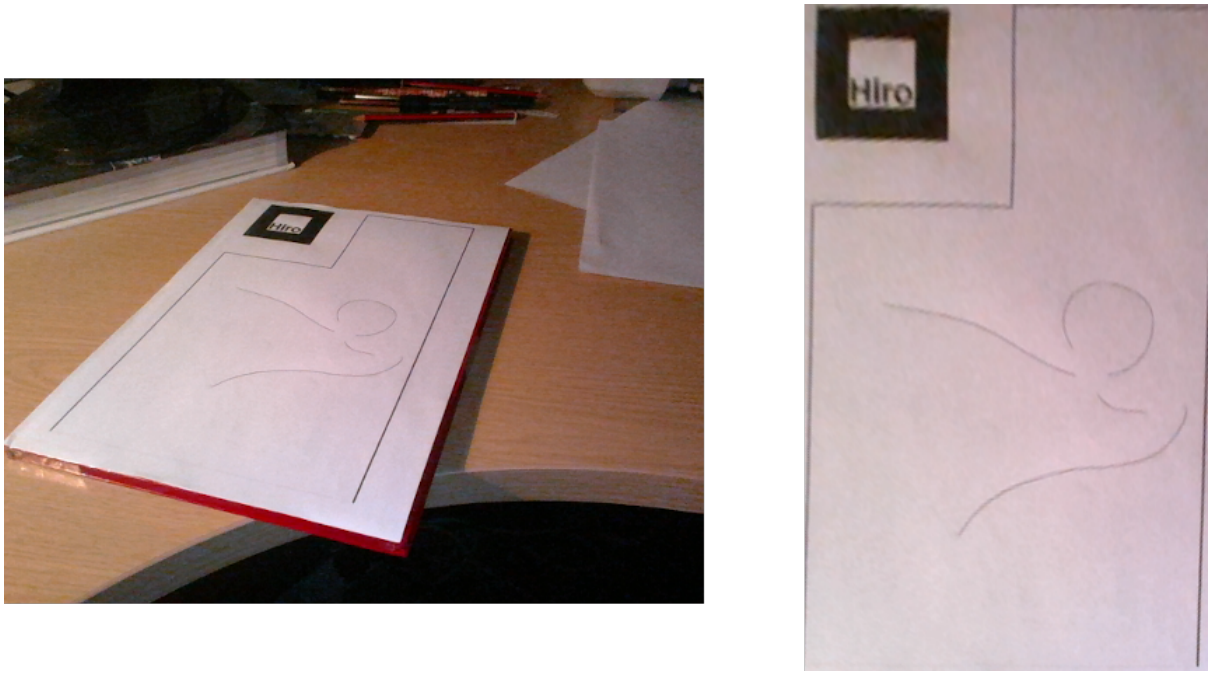

Figure 4.1: *Real-world testing of the perspective transform with OpenCV. Left: original photo of the page from an angle. Right: warped image using the perspective transform with OpenCV*

theirs screens. These scenarios needed to be allowed for when projecting the points on the video stream frames and so a dynamic resizing method was implemented, based on the live screen dimensions. The debugging methods listed in Section 4.4 were useful in finding the correct parameters.

4.3 Line detection and extraction

This part of the project used image processing to identify and extract the track lines into a suitable vector for the game engine. This experimental process was very time consuming and required many tests involving multiple permutations of algorithms, transforms, and their parameters. Following is a list of most of the tested image processing methods that were used in isolation and in combination with each other:

- grayscale conversion
- box blur, median blur, Gaussian blur, bilateral blur
- binary thresholding
- Sobel filter, Laplacian filter
- Canny edge detection
- topological/morphological skeleton

- erosion, dilation
- contour detection and approximate polygon contours
- line segment detector (LSD) [11]
- Hough transform and probabilistic Hough transform
- unsharp masking.

Most of the experimentation involved a grayscale conversion and a blur. The Hough transform and LSD extracted the straight lines but failed to extract the curved parts of the track. The Canny edge detector extracted most lines but showed breaks and patches in the lines as seen in Figure 4.2. This was fixed by applying an image dilation afterwards to merge any broken lines. The contour detection method was tested without previous Canny edge detection but this resulted in inaccurate line detection. Contour detection was also performed without subsequent polygon approximation, but this resulted in jagged lines as shown in Figure 4.2 and too much detail for the game engine.

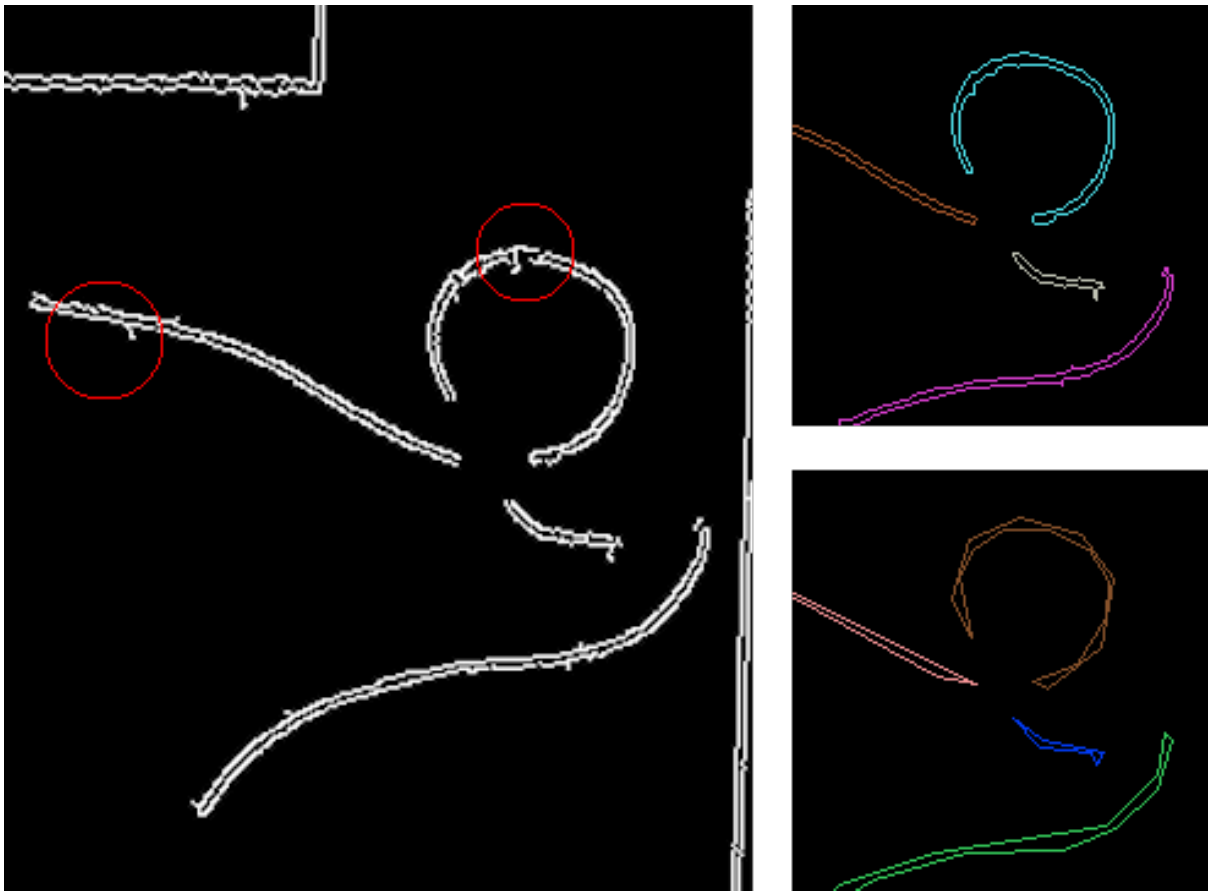

Figure 4.2: *Issues encountered with image processing. Left: output of raw Canny edge detection with line breaks highlighted. Right: comparison of contour detection without polygon approximation (top) versus with polygon approximation (bottom).*

The final sequence of image filters and transforms, as specified in Section 3.3, are shown visually in Figure 4.3 from video capture to final line detection.

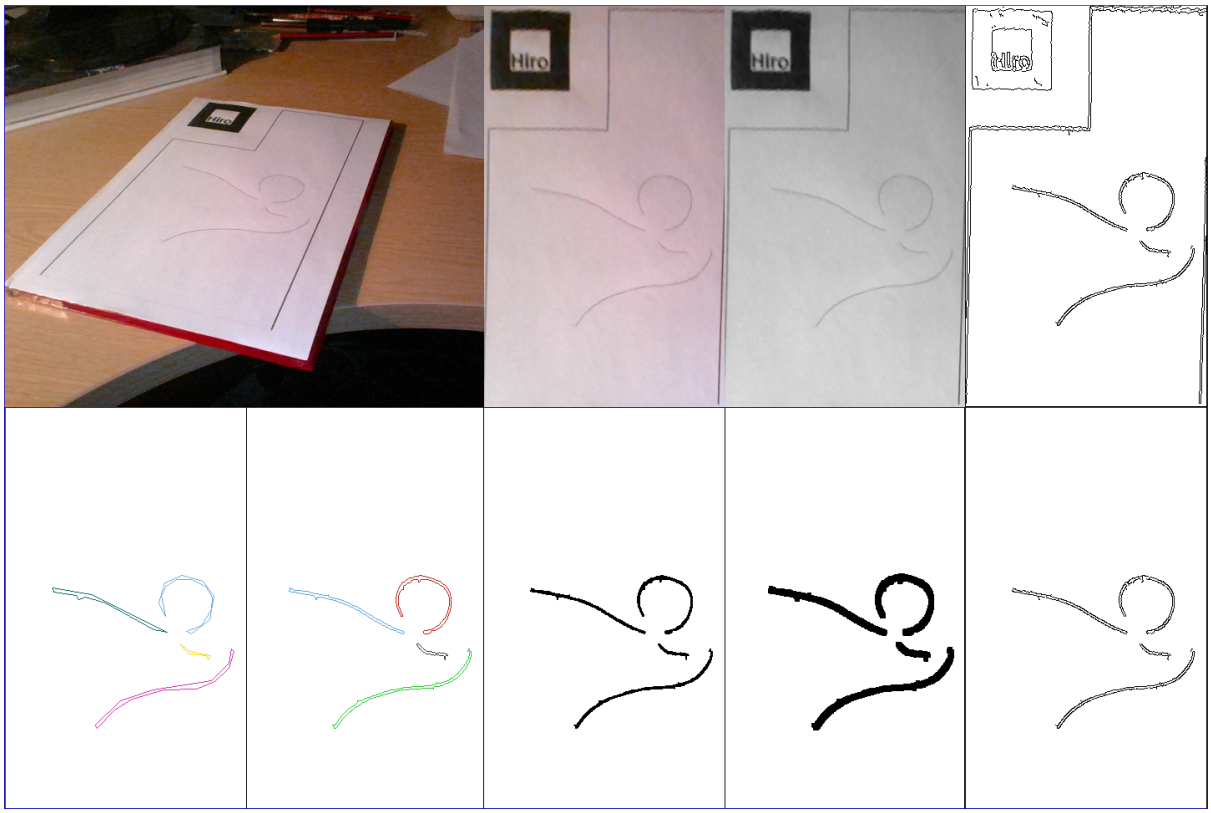

Figure 4.3: *Example of real image processing sequence. From top-left clockwise: original video, perspective transform, grayscale conversion, Canny edge detection, marker/border masking, dilation, erosion, contours, approximate polygons contours.*

4.4 Debugging

Several visual methods of debugging were used to assist in the experimental stage of the project, and are shown in Figure 4.4. The corners of the detected drawing area were overlaid on the video stream to ensure that the marker detection and coordinate conversion was working correctly. A virtual plane object was rendered in the frame to further help in identifying any problems with page localisation. A third method used for testing was a live stream of the image processing results, appended to the video stream. This allowed viewing of the contour detection and line extraction in real-time, making it easier to adjust parameters and see quick results.

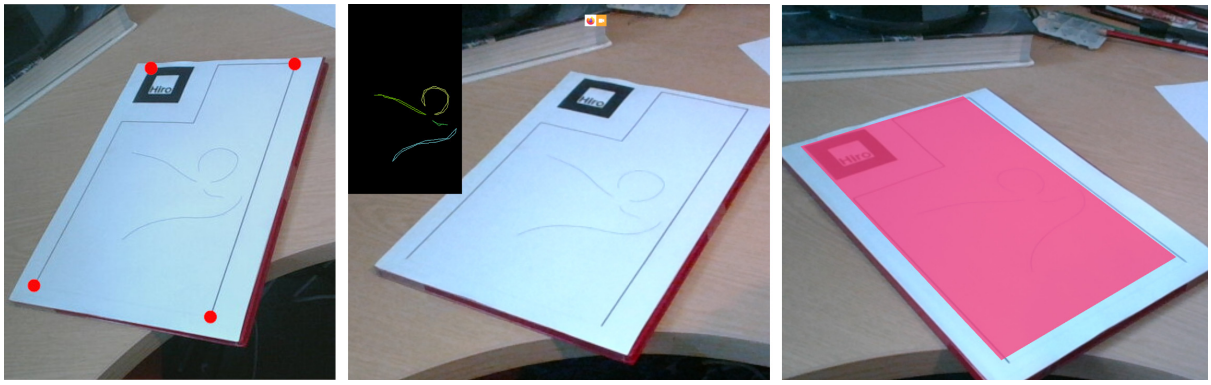

Figure 4.4: *Examples of some visual debugging methods. Left: corner marking. Middle: live image processing view. Right: overlaid plane on drawing area*

4.5 Simulated track results

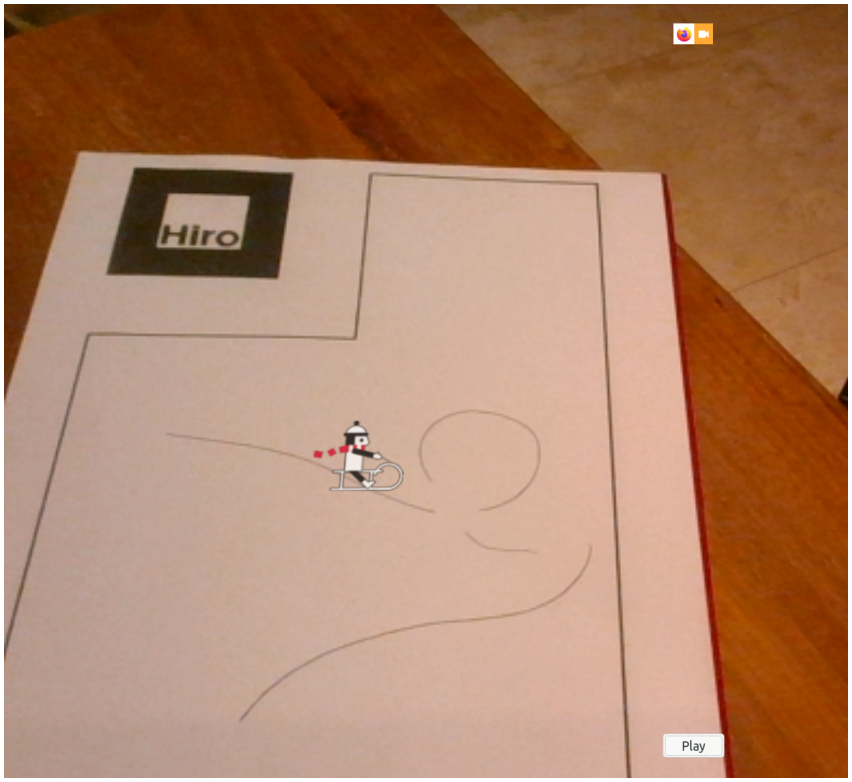

Figure 4.5: *Screenshot from the working game prototype*

Chapter 5

Analysis and Discussion

The perspective transform functionality worked well to warp the video stream, however it may be more accurate to simply take a frame from the video capture as before, but instead of using the THREE AR to continuously track the marker the detection would be performed on the still image using another library. This would yield more accurate results as it wouldn't fall victim to the tracking stabilisation which smooths the movement rather than achieve perfect tracking. Another potential improvement for a better perspective transform would be to warp the image until the marker is detected to be perfectly square. This would ensure that the warped image results in the most parallel angle to the drawing surface as possible.

In the image processing stage it may be interesting to see the effect of performing the perspective transform last, on the contour vectors, having performed all the other image processing beforehand. This may preserve the detail in the original image, instead of potentially introducing artefacts due to image warping and resizing. This would also be marginally quicker as only the contour points would need transforming, not the whole image.

Within the image processing pipeline, one method that wasn't thoroughly tested was morphological skeletonisation. This was due to the lack of available JavaScript libraries for this method. This would be an ideal replacement for several steps of the current pipeline as it performs roughly the same functionality as the image dilation and erosion steps. It also reduces the width of the lines automatically down to the narrowest width which would improve line fidelity and extraction. Another method which could be further explored is the Laplacian filter which yields results similar to the skeletonisation methods. The only issue with the Laplacian filter was that the lines were more patchy in places and had some considerable gaps. This could possibly be reduced with further experimentation.

Chapter 6

Conclusions

6.1 Summary

The project was a success and resulted in a working prototype of the game Line Rider in augmented reality, as seen in Figure 4.5. The project objectives were achieved and the game is now playable on any modern device with a camera and a web browser, without the need for a server. The lines are extracted accurately and the official Line Rider physics engine was inserted into the AR game pipeline to yield authentic game simulations.

6.2 Limitations and future improvements

The overall stability of the overlaid game objects is good, but there is some room for improvement. The tests were performed on a five year old iPhone 7 so the game performance and marker detection could be better on a newer and faster device. The marker tracking could also be optimised for smoothness in movement, rather than perfect localisation. The line detection is accurate however the track is still represented as a collection of polygon contours which results in some small bumps. A solution to this would be to implement curved lines, which can then be fed into the game engine. This would lead to a more accurate depiction of the track in the physics engine an a smoother ride. A drawn circle could then be accurately depicted as a curve instead of an array of small angle line segments.

References

[1] R. Cabello. Three.js. https://github.com/mrdoob/three.js, 2010. [Online; accessed 31-May-2021].

[2] J. L. Milner. THREE AR. https://github.com/JamesLMilner/THREEAR, 2019. [Online; accessed 31-May-2021].

[3] S. Taheri, C. Pan, G. Song, W. Gan, M. R. Haghighat, and N. Hu. OpenCV.js. https://github.com/ucisysarch/opencvjs, 2017. [Online; accessed 31-May-2021].

[4] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973.

[5] D. Lu and M. Henry. Line Rider Core. https://github.com/conundrumer/lr-core, 2016. [Online; accessed 31-May-2021].

[6] J. Garnier. Anime.js. https://github.com/juliangarnier/anime, 2016. [Online; accessed 31-May-2021].

[7] E. Wallace and D. G. Taylor. glfx.js. https://github.com/evanw/glfx.js, 2011. [Online; accessed 31-May-2021].

[8] E. Zatepyakin. JSFeat. https://github.com/inspirit/jsfeat, 2012. [Online; accessed 31-May-2021].

[9] D. Marcos. A-Frame. https://github.com/aframevr/aframe, 2015. [Online; accessed 31-May-2021].

[10] J. Etienne. AR.js. https://github.com/AR-js-org/AR.js, 2020. [Online; accessed 31-May-2021].

[11] Rafael Grompone von Gioi, Jérémie Jakubowicz, Jean-Michel Morel, and Gregory Randall. LSD: a Line Segment Detector. *Image Processing On Line*, 2:35–55, 2012. https://doi.org/10.5201/ipol.2012.gjmr-lsd.