Autonomous Driving in Mario Kart: Double Dash

James Hudson

***Abstract* - A system that uses footage from Mario Kart: Double Dash and processes in real time to make decisions about how to navigate a course. This system is built with a defined goal to finish a Time Trial on the course "Baby Park". This paper includes methodologies for making driving decisions in real time using several Image Processing techniques. I hope that this paper serves as a proof of concept for implementing these ideas in real-life driving.**

I. Introduction

Autonomous Driving is becoming a very heavily researched topic with many resources being put towards achieving it in the most efficient and safe manner possible. Many state of the art technologies use Neural Nets and Deep Learning in order to make the best decisions possible when driving.

As an introduction to Autonomous Driving, this system uses much simpler methodologies that are not as expandable to broader use cases. This system is merely used a proof of concept that these simple techniques can be used to achieve a very base level of Autonomous Driving.

This system will be used to prove that a combination of very simple Image Processing techniques can be used in unison to achieve results in Autonomous Driving.

II. Problem Definition & Methods

II.I. Problem Statement

As mentioned above, this system will be used in a very isolated environment which is a Time Trial in Mario Kart: Double Dash on the course "Baby Park". This provides a much more limited set of variables to consider when designing Autonomous Driving. In this situation, we have many constants such as the color of the track, field of view of the camera and areas to be masked out of view.

II.II. Image Processing

Implementing the system begins with capturing footage of the video game which is handled using OpenCV's ImageGrab functionality for every frame that the game is running. This provides us with an image which we can process. The following are pre-processing to prepare the image for further algorithms. Upon retrieval of this image, we apply a Gaussian Blur to eliminate any noise that may be present. Following this we want to filter the image such that we are only handling the portions of the image that contain the track. This was solved by converting the image to HSV and defining ranges of colors that would represent those in the track and filtering the image to only contain these colors. Given the fact that our field of view is constant, we can mask the image to the common area in which we find the track. This is done to eliminate any false positives found from anything in the image that is not relevant to the track (sky boxes, background animations, etc.). Finally, we apply dilation to further reduce noise to provide the cleanest image possible for the next step.

Given our pre-processed image we can now apply Canny Edge Detection for detecting the edges of the road. Since we are using a masked image, the most pronounced lines detected in the image are the left and right edges of the track. Finally, this edge map is used in the Hough Line Transform that outputs the lines located at the left and right edges of the track. For visual reference, we show these lines on top of the input image.

II.III. Handling User Input (Driving)

The output from above (lines representing the left and right edges of the road) is now used to calculate how to handle steering and acceleration. The line angles will be measured from 0° to 180°

with 0° being a line pointing to the left and 180° being a line pointing to the right; to eliminate further false positives, we will separate this into 5° increments for which we will only store one line for each increment. This means that for any given frame we will only be handling at most 36 lines. At its core, we have 3 states which will be determined by the lines that we detect in any given frame. Once a state is determined we use user32.dll to handle user inputs to the emulator.

The first state that we can consider is that in which we are detect a straight track segment. This is a trivial state to detect. We first want to get an average of all line angles detected.

$$angleAvg = \frac{\sum(angles\ of\ all\ lines)}{\#\ of\ lines}$$

This state is determined when angleAvg is calculated to be between 80° and 100°. This gives a threshold which represents a straight road. If this state is determined, we input acceleration without any turning.

The second state that we can consider is that in which we detect a turn. This is also a trivial state. As with the previous state, we calculate angleAvg. There are two possible sub states now: a left turn and a right turn. We determine a left turn if:

$$0° < \text{angleAvg} < 70$$

Given this, we input acceleration and a full turn left. Otherwise, we determine a right turn if:

$$110° < \text{angleAvg} < 180°$$

Given this, we input acceleration and a full turn right.

Finally, the last state that we detect is that in which we detect we have run into a wall. This state is less trivial to detect than the previous two. Detecting this state involves calculating the angleAvg as with the previous states. Upon trial and error, it was determined that this state could be detected if:

$$\#\ of\ lines < 2$$

This alone can lead to many false positives. For example: when we only detect 1 edge of the road on a straight segment because of noise. To solve this issue, we keep a count of how many consecutive frames there have been less than 2 lines detected. When this count exceeds 10 frames, we can confirm that the state has been detected.

As with the previous state, we have two sub states: running into the left wall and running into the right wall. These are important to differentiate as we will want to reverse in different directions to avoid facing the wrong way. We determine that we have run into the right wall if:

$$angleAvg < 90°$$

Given this, we input acceleration in reverse and a full right turn for a duration of 0.5 seconds. This value was determined through trial and error. Otherwise, we can determine that we have run into the left wall if:

$$angleAvg > 90°$$

Given this, we input acceleration in reverse and a full left turn for a duration of 0.5 seconds.

III. Results and Discussion

The results of this system proved to be successful for its intended purpose. The system was successfully able to use all the methods defined above to determine the edges of the track and make a driving decision based on which case is detected from the previous section.

However, this system is limited in several areas which – in turn – limit its expandability to further areas. Many of the methods which are used were tuned to work only on a given track and would need an overhaul to be used more broadly. The most prevalent method would be the HSV color filtering that we use in attempts to only consider the track when doing Canny Edge Detection. The filtered values are hard coded for the track which we use for testing which does not allow for this system to work on other tracks.

The major shortcoming of this system is its ability to detect turns. As it stands, most of the turns that it inputs tend to not be tight enough and we end up running into the outside (left) wall. I believe that this could be solved with an analog input system to handle turning as right now we are using a digital left and right system. Given this we would have much more fine control with regards to turning and would be able to better emulate a human-like turn. Furthermore, it would be beneficial to tune the parameters passed to the Hough Line Transform to handle detected edges that are slightly curved during

a turn. I believe that these two changes combined would allow the system to negate running into walls during turns.

IV. Conclusion

Development of this system allowed to have hands on experience with many different Image Processing techniques and visualized how they can be applied together to reach an end goal. There are many areas in which this system can be improved such that its use case becomes broader.

Systems such as these have potential to be pushed beyond video games and into real-world applications. However, environments such as video games provide the ideal environment for initially building these systems. We can emulate many of the hazards that driving poses in the real world without the dangers that would come with testing in the real world. It is to be noted that the environment which we tested in provides many less hazards than that of a video game which is attempting to mimic real life driving. This is a very simple base case where our only goal is to finish a Time Trial. Real-world applications would have goals more tailored to safety and practicality (avoiding obstacles, refined turning, etc.) and would ideally have a Neural Net or some learning algorithm which allows it to know whether or not the decisions it is making are good or not.