Game AI Project Report

Title of Project: Magnet Racers - AI Implementation

Team:

Team BABU

Authors:

Absinthe (Mengling) Wu

Benjamin Dunbar

Uttkarsh Narayan

William Wisheart

CS 4150/6150: Game AI

Magy Seif El-Nasr

TA: Zhengxing Chen

April 18, 2017

Project Team Plan

Our team was unique due to the fact that we had split backgrounds between the four members. Ben and Bill are both undergraduate Computer Engineering students with a good amount of programming experience between the two. Meanwhile, Absinthe and Uttkarsh are both graduate Game Design students with less programming experience before this class. However, both Absinthe and Uttkarsh have Unity experience and were involved with the creation of Magnet Racers which gave them insight on how to implement the AIs.

Ben: Ben was the team leader for the project. Primary contributions included heavy refactoring of the c# code as well as creating the Q-Learning AI. Ben also assisted with writing and presentation slides.
Bill: Bill's primary contributions to the team were creating the Decision Tree AI as well as writing the Final Report. Bill also assisted with refactoring the game's code, graphing test data, and creating the presentation.
Absinthe: One of the original creators of Magnet Racers. Absinthe also assisted with writing the proposal for the project as well as the presentation. She created the video demos for the project and worked on UI updates to the game in Unity as well as testing result data.
Uttkarsh: One of the original creators of Magnet Racers. Uttkarsh's primary role was assisting Ben with creating the Q-Learning bot as well as adding complexity to the Decision Tree AI. Uttkarsh also assisted with writing the proposal and Final Report.

Vision Statement

The goal of this project was to apply concepts learned throughout this semester by introducing AI agents to the game Magnet Racers (developed by ATSU production company during the previous semester). Magnet Racers is a game in which 2 to 4 human players race magnets around a track. Each player can manipulate his or her magnet by toggling their magnet's polarity. This will cause the magnet to interact with fixed magnets on the map and move from point A to B and so on. Additionally, the magnets of each of the players will attract and repel each other.

To make the game more interesting we decided to implement AI concepts to challenge human players. The first AI agent implemented is a random move AI to give a baseline for both of the following AI as well as the human player. The next AI agent is a hard coded Decision Tree AI. This agent uses a preset Decision Tree developed from play testing the game and determining the tactics the bot should use. Finally, the last AI agent implemented is a Q-learning bot. This bot uses a reward system to find an optimal path around the track. Together, all three of these agents will be able to give a new player a competitive racing experience.

Details about the game can be found on- https://www.uttkarsh-narayan.com/magnet-racers

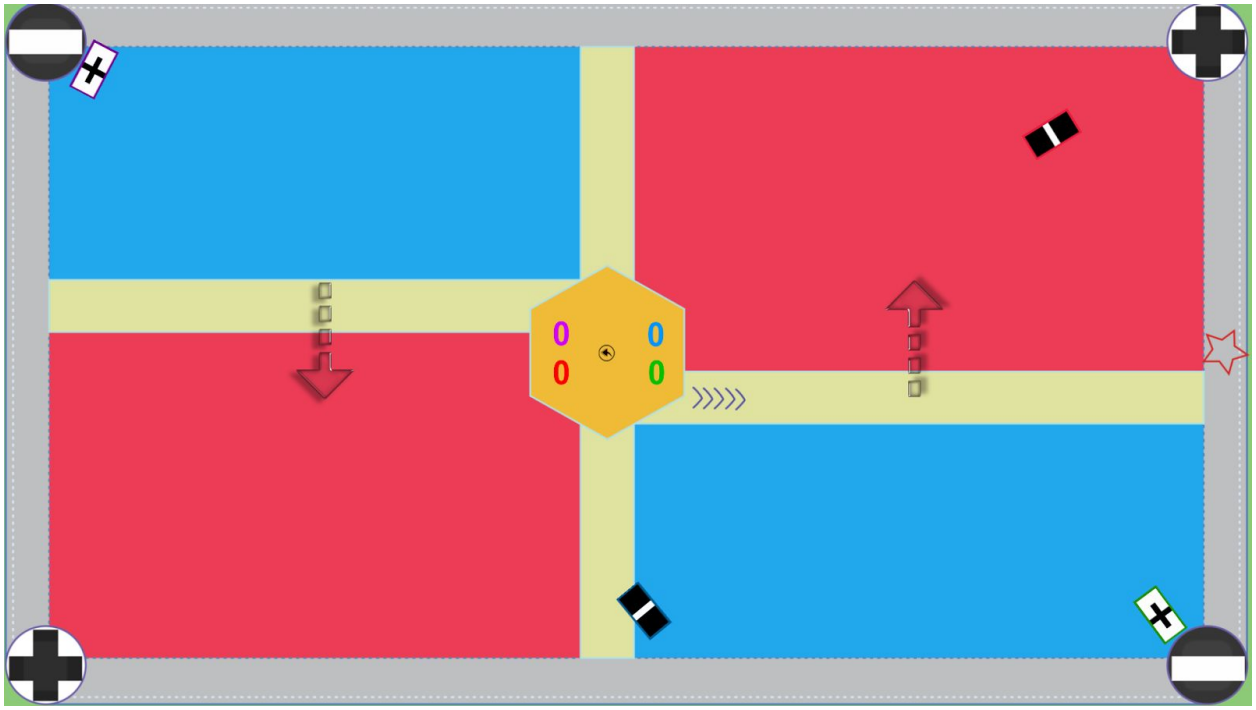


Figure 1: Visual of existing Magnet Racer game

System Design/Architecture

Magnet Racers was designed as a single scene Unity game. The scene contains an image background as seen in Figure 1. The scene is then populated with Game Objects to complete the track. Four large fixed magnet objects called ‘poles’ are placed into the corners with alternating polarities. These provide a propelling/repelling force on the racers to push them around the map. Four Gate Trigger objects separate each quadrant of the map. These gates are invisible to the player but are used to ensure the racers are proceeding through the game in the correct direction. The final four game objects are the magnet objects controlled by each racer.

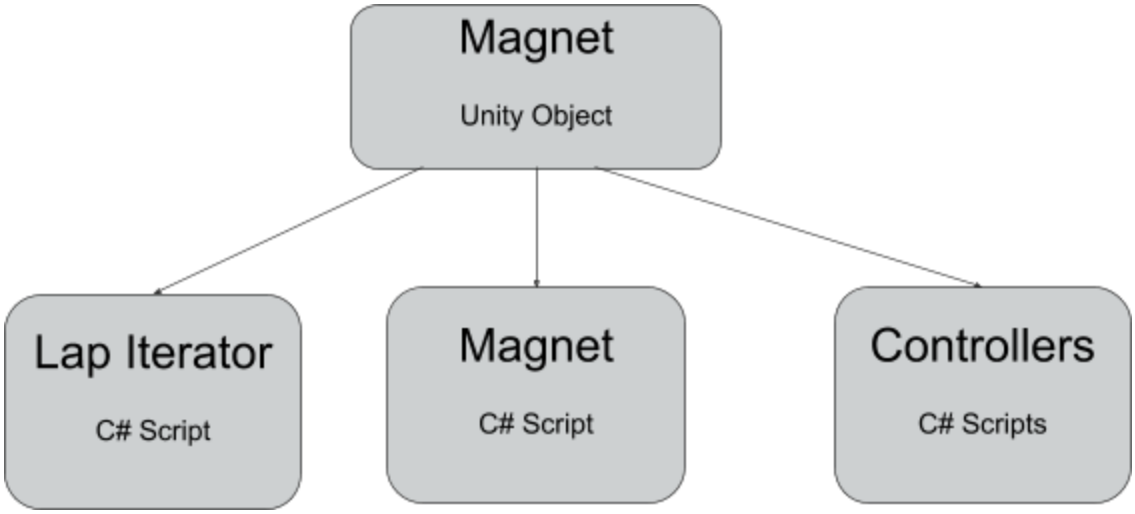


Figure 2: Magnet Game Object Components

The Magnet Game Objects contain three script components outlined in Figure 2. The first script is the Lap Iterator script. Lap Iterator keeps track of multiple game scenarios for the individual racers. The script keeps track of the next gate the racer needs to pass through and iterates the racers lap count if the gate was the last in the lap. If the racers lap number equals the win lap count then Lap Iterator is tasked with signaling to the game controller that this racer has completed the race. On top of all this, Lap Iterator can be set to keep track of and record lap times to a text file. This is useful for the Q-Learning agent which will be discussed later on.

The second script contained by each Magnet Game Object is the Magnet script.This script contains the characteristics of each magnet including the magnet charge as well as repel strength. On startup this script also renders the magnet racers sprite. Every 0.02 seconds the FixedUpdate function of this script is called. FixedUpdate calculates the forces acting on the racer from the poles as well as the other racers and updates the magnets sprite accordingly.

The last scripts attached to the Magnet Game Objects are the controller scripts. The primary function of the controller scripts are to determine when to toggle the magnets charge. The first three controllers in Figure 3 below are the AI agents implemented in this project and will be discussed further below. The last controller, the Human Controller, allows for a key to be assigned to the magnet and each time the key is pressed the magnets polarity is toggled. For debugging purposes the Human Controller script was attached to the three AI controlled magnets as well.

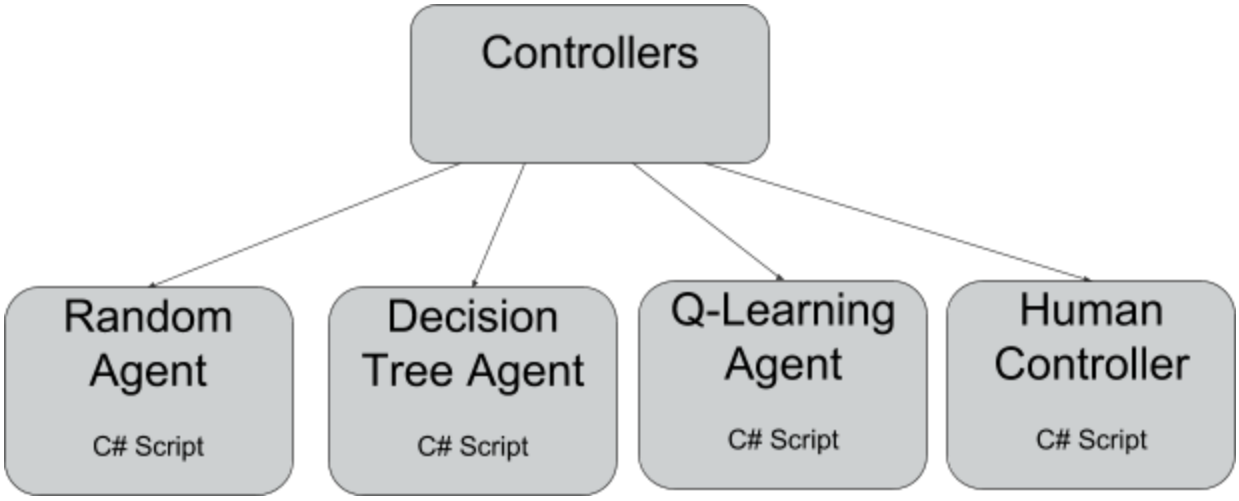


Figure 3: Magnet Controller Scripts

Random AI Agent

The Random AI Agent uses a simple a simple random number generator to determine whether to toggle the magnets polarity. The RandomAgent script contains a FixedUpdate() function that is called every 0.02 seconds, or 50 times a second. If a randomly generated value between 0 and 1 is less than 0.02 the magnets polarity is toggled. This results in the magnets polarity being toggle approximately once every second and allows the magnet to slowly make its way through the track.

Decision Tree Agent

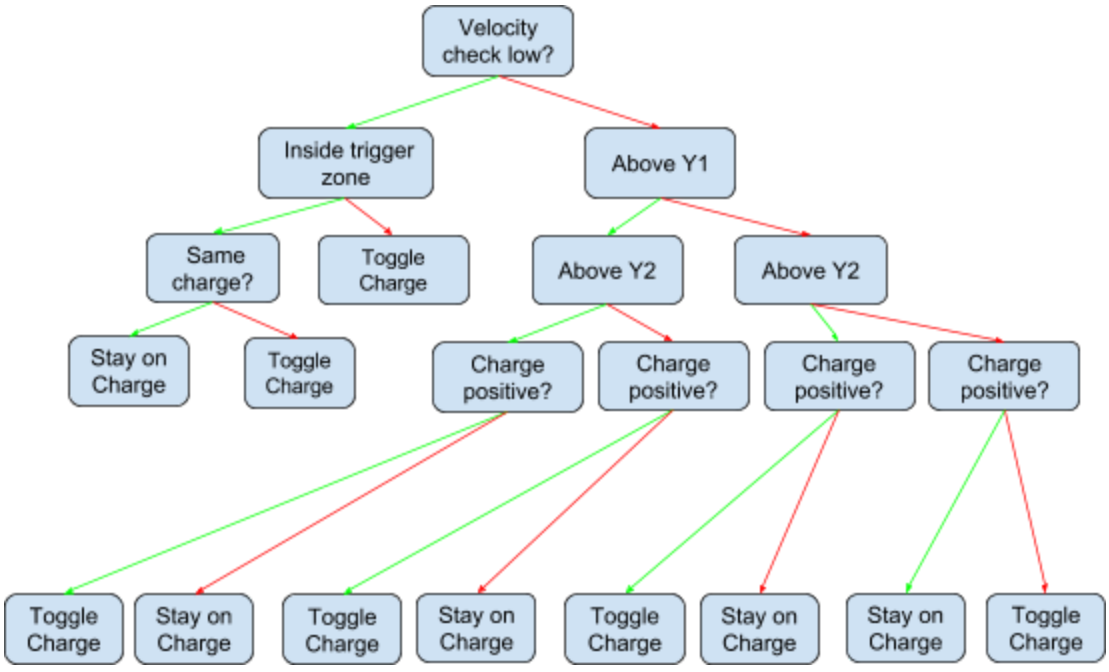


Figure 4: Decision Tree Diagram

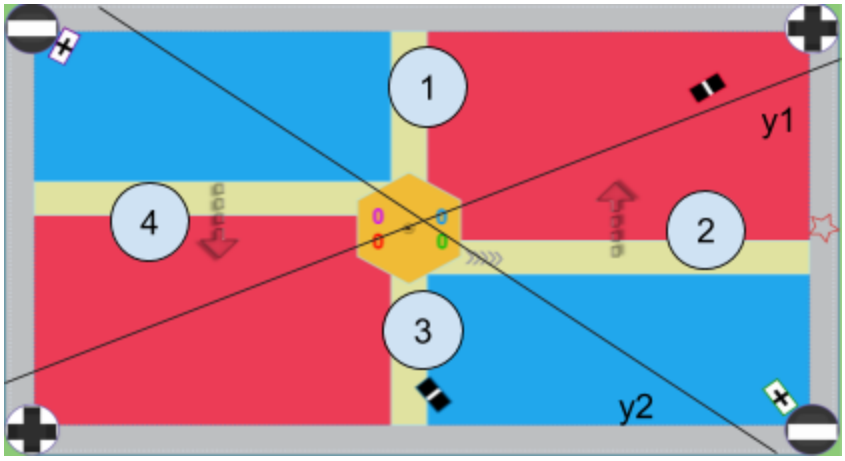


Figure 5: Decision Tree Quadrants

The Decision Tree agent uses a hardcoded Decision Tree as seen in Figure 4 above. Similarly to the Random Agent, the DecisionTreeAgent script contains a FixedUpdate() function that runs every 0.02 seconds and checks the magnets current state to the tree. Through play testing we found that a simple strategy to populate the tree diagram and propel the magnet around the track was to section off the track into four quadrants as seen in Figure 5. Based on which quadrant the magnet is in as well as its current charge, the bot is able to toggle its charge and build momentum in the correct direction. To add more logic to the DT AI agent we added a check for the magnets velocity. If the magnets velocity dipped below a certain speed, likely meaning the magnet is stuck in a corner, the bot will toggle the charge and push itself away from the wall.

Q-Learning Agent

The Q Learning agent is composed of 3 different components: QState, QMap and QLearningAgent. The QState and QMap components make the functionality of the Q learning agent more clean and manageable. The QState component gives developers an easy way to summarize and calculate various game states needed for the Q-learning agent. Here the team calculated and stored the current magnet's charge and its position as a quadrant (similar to the quadrants used in the decision tree). The state also contains the charge, direction, and distance of the closest opponent so that Q-learning agents can react to opponents. Within the QState the parameters are all succinctly stored as int values, and when QState is written as a string they are concatenated with ";" and put in the order: thisCharge, thisQuadrant, nearestOpponentDistance, nearestOpponentDirection, nearestOpponentCharge. The opponent direction is summarized to up, left, down, right (0,1,2,3) and the distance is reduced to near (1) and far (2) in order to give the most brief representation of how opponents may affect the user.

The QMap component holds all of the learned data and offers reading/saving functionality with the text files. It stores this data in a Dictionary of QState keys and int[] values in the class and offers convenient functions to the QLearningAgent which will have a QMap member. An example of a learned QMap is provided in Appendix A and shows a list of QStates represented as 5 numbers delimited with semicolons. On each line after the state are two integers, which are the Q values for the toggle and maintain charge actions.

The QLearningAgent script is the actual agent in control of what action a Q-learning magnet takes. The QLearningAgent script contains the code where the reward is calculated and where a new Qvalue is calculated and updated after an action is chosen from a game state. The QLearningAgent offers up a function that can be called each time a lap is passed, so that it can delegate the QMap to store the current data to a text file. The file location and base fileName are public variables which can be set from the Unity editor. In FixedUpdate(), which runs every physics update of the unity game. The QLearningAgent calculates the game state with a static function offered by QState. Then it calculates a reward and updates the Qvalue corresponding to the last state and last action. It will then chose an action and update the memory variables.

QLearningAgent offers three types of different reward systems: gate rewards, motion rewards and force rewards. Magnet racers was really difficult to optimize a Q-learning algorithm for because of the chaotic nature of the game. Even with the fairly thorough state summary, the rewards had to be worked on in depth to get a working agent. Just using gate rewards, giving the agent a reward every time it enters a different quadrant, weren't sufficient because the magnet often has momentum and could still pass the gate under circumstances where the charge wasn't desired for that state. To improve upon this, another reward system was created which constantly gave rewards if the magnet was following a counterclockwise motion, but this also had some inherent flaws because a magnet once again could move in the right direction with the wrong charge because of momentum. Finally, a last reward system was created which would analyze the actual forces on the magnet. This was the best method because it also allowed for quick rewards based on the influence of opponent magnets.

In terms of calculating a new q value, the QLearningAgent implements the algorithm:

Q(s,a) = Q(s,a) + learning_rate(reward + discount_factor * Max(Q(s',a)) - Q(s,a)). This allows it to update q values not only based on rewards, but also on the existing value and the future possibilities from the state it ended up in. Because of the sheer speed of the game (hundreds of choices or more a lap), a relatively low learning_rate was used so that the q value wouldn't vary too quickly and be ineffective.

While choosing the new actions, the Q-learning agent would use the maturity of the learning map to decide whether is should explore like a random agent, or whether it should usually choose the action which it has learned to be effective. In our case, the 'age' or maturity of the algorithm was incremented every lap. However, since an agent made so many actions each lap, the fixed learning maturity could be a low value. The team settled on 2 laps of complete random exploring, which could allow the agent to enter hundreds or even thousands of states as a random agent. When the algorithm was mature, it slowly explored less often, but

still had a minimum exploring rate so that it could get out of situations where it was stuck in a corner.

After testing, the Q-learning agents which relied on motion and forces for rewards both performed quite well, even better than the decision tree agent. As seen in the results section, a game was played which was modified to allow for 40 laps and times were recorded for each of four agents: random, decision tree, q learning based on motion rewards, and q learning based on force rewards. The winner was the q-learning algorithm based on force rewards. It could offer a challenge even to a human player.

Tools and Libraries

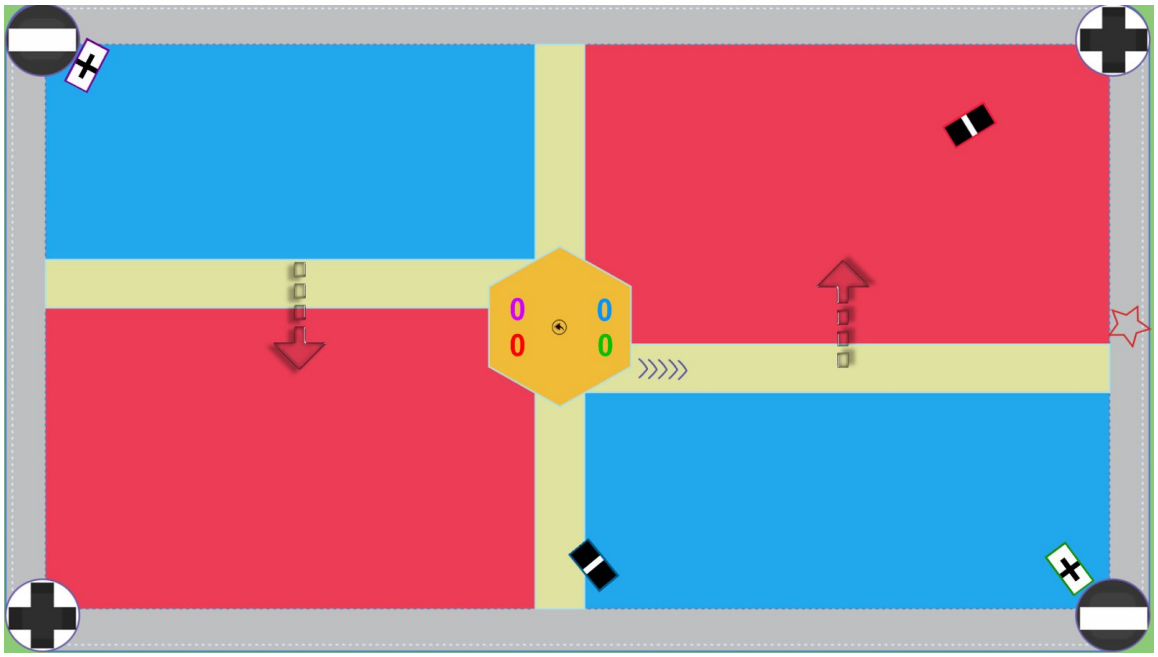
Engine: Unity 5.5

The original alpha version of Magnet Racer was developed in Unity 5, we decided to keep the same game engine so that the development of the game will not hamper the team with time constraints. C# has been chosen as the main programming language because it is very compatible with Unity 5. The team will therefor be using C# when implementing each of the AI agents to the game.

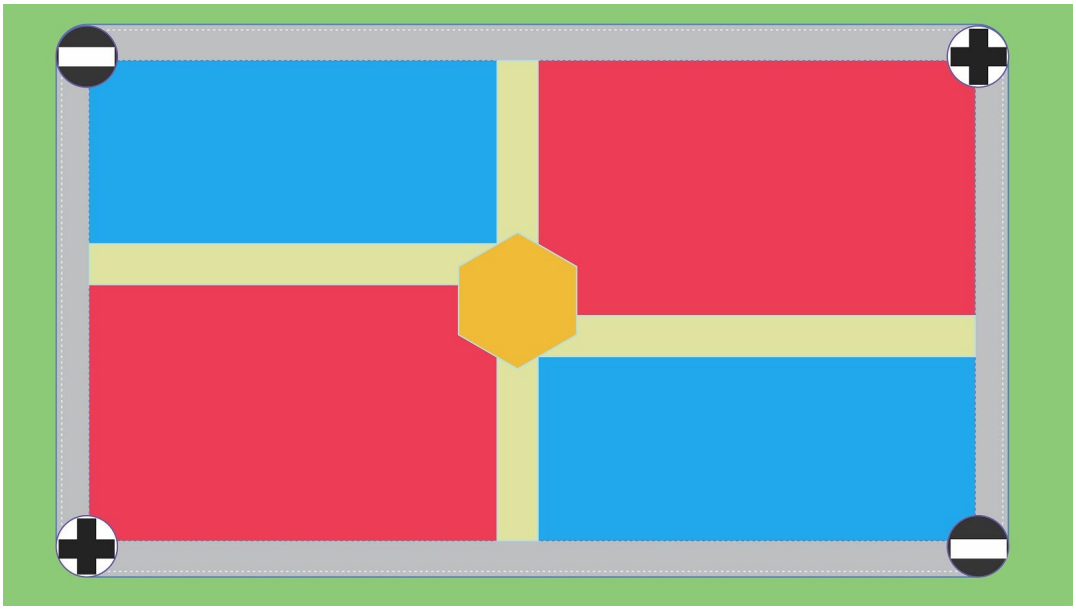
Language: C#
- Very compatible with Unity 5.5
- Existing scripts are in C#

Sharing: Github
- Allow for collaboration on different computers

Magnet Racer level:



Videos



Game Demo, Click this Image to play the game Demo

The team has created game play videos of a random agent, decision tree agent, and humans versus random and decision tree agents. The team has also created a video with four agents competing: random, decision tree, q learning based on motion rewards and q learning based on force rewards. For the four agents competing there is a video while the q learning agents are still in a strict learning phase and a video where the agents are in an online learning phase after the fixed initial learning is complete.

In the learning and bots competition portions of the video the bots are as follows:

- Red: random
- Purple: decision tree
- Green: Q-learning - Force Rewards
- Blue: Q-learning - Motion Rewards

In the full game video, two humans are shown controlling Green and Blue.

Results

The team has also been able to record timed data on the agents as they compete against each other during a 40 lap game. The team has also compared the averages to a human average. The result of the project is that it is now possible to have an engaging game against other humans and AI agents at the same time, or even just compete with AI.

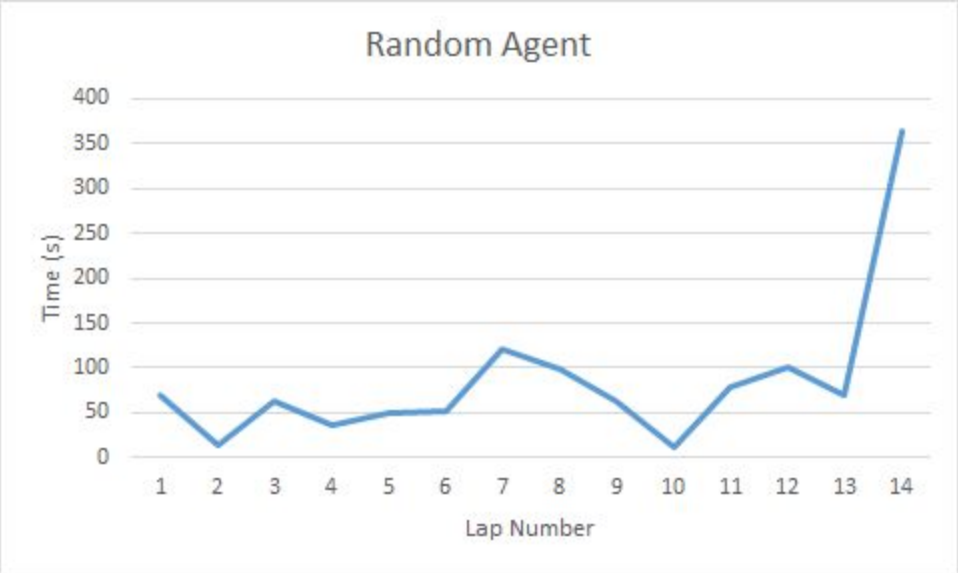


Figure 6: Random Agent Lap Times

The average Random Agent lap time came in at 85.4 seconds. Due to the fact that the other racers finished, thus ending the race, the Random Agent was only able to complete 14 laps in a 40 lap race. The Random agent also noticed an increase in time when it was on the lap alone because other magnets wouldn't pull it along as often.

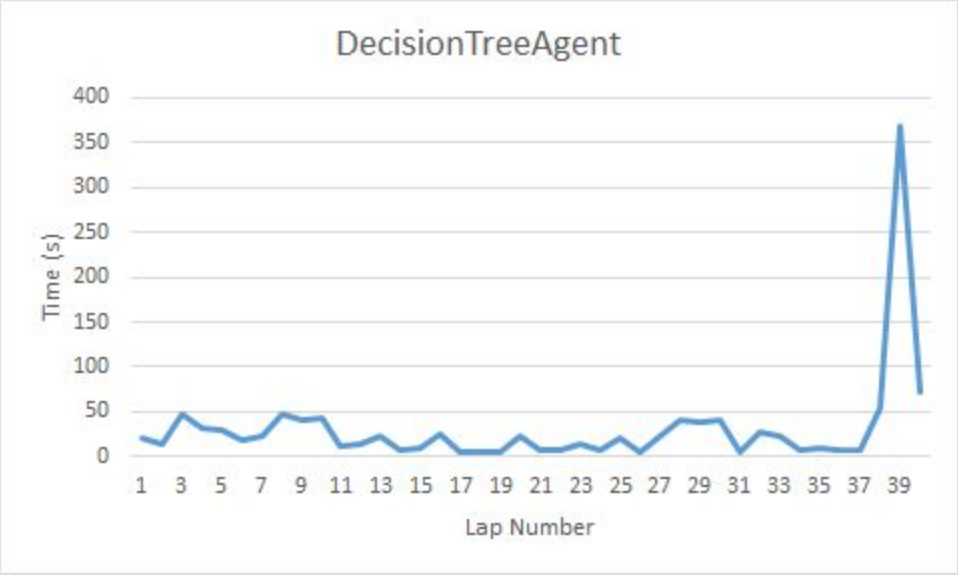


Figure 7: Decision Tree Agent Lap Times

The Decision Tree Agent's average lap time in a 40 lap race was 30.81 seconds. Due to challenges with the decision tree algorithm, the decision tree racer sometimes relied on the other magnets to dislodge it when it was trapped in a corner. If not for the last 3 laps, where the

two q-learning racers were no longer racing and the decision tree agent was stuck more often, the decision tree agent averaged a competitive 19.9 seconds.

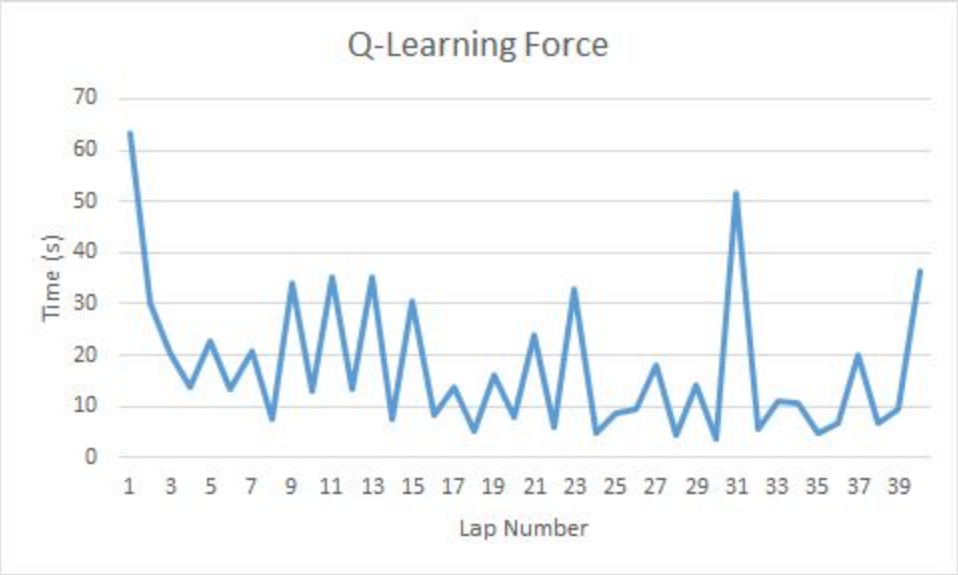


Figure 8: Q-Learning Force Agent Lap Times

The Q-Learning Force agent's average lap time in a 40 lap race was 17.55 seconds. This was the most competitive AI racer overall.

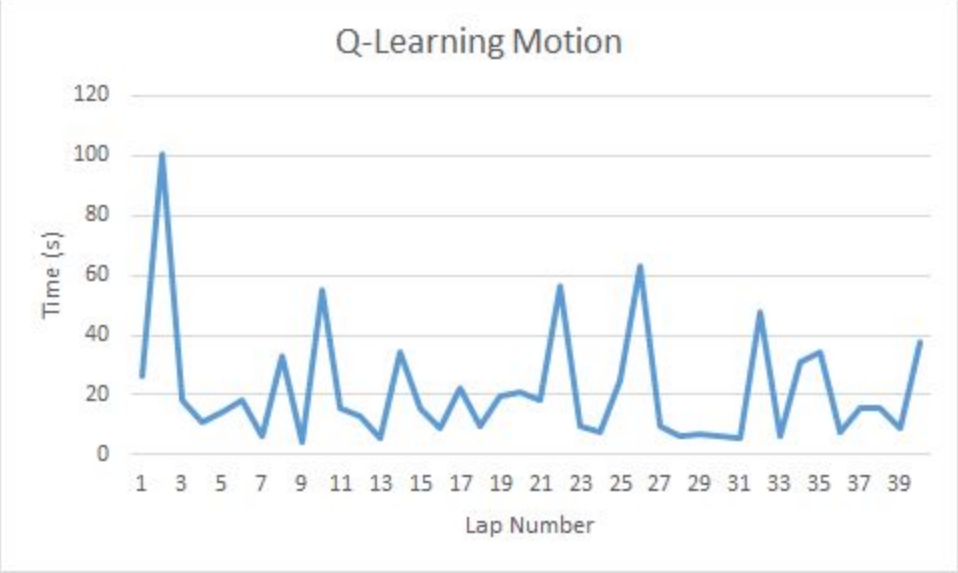


Figure 9: Q-Learning Motion Agent Lap Times

The Q-Learning Motion Agent's average lap time in a 40 lap race was 21.78 seconds. It was a decent agent, but not the most impressive.

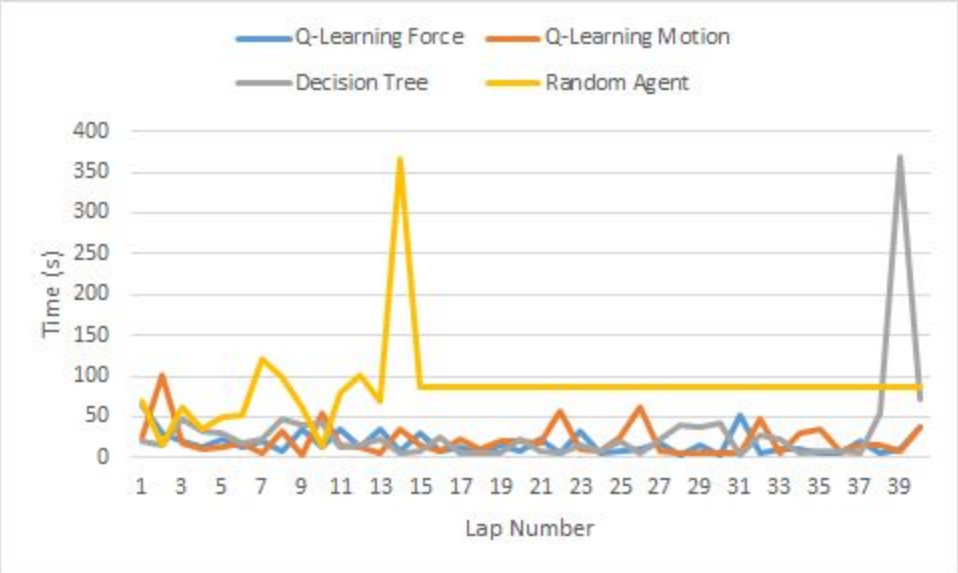


Figure 10: Four Agent Lap Times

The figure above compares all four AI Agents lap times from the same race against each other. As you can see, the random agent performed the worst overall, with the decision tree in second place. The Q-learning agents had a few slow laps during the fixed learning period, but then did better than the decision tree agent on average. They were able to make up for their slow first runs and come in first place (Force reward agent) and second place (Motion reward agent).

One thing to note about the magnet racer game is the sheer chaotic nature of gameplay. Even after the learning algorithms reached a more steady part of their learning curve, they had a large standard deviation. However, the chaotic movement and momentum in the game also affects human players, so the AI is difficult to troubleshoot, but not at a large disadvantage in terms of possible success. The best AI agent had an average time of 17.55 seconds, which was quite proficient and can beat humans depending on skill level. In a 10 lap run with 2 humans and two Q-learning bots, it was determined that a human player has an average between 16.7 and 29.8 seconds per lap. This 10 lap game between two humans and two bots is recorded in the final graph in figure 11 below. The final game in question had the first human as the winner and a pre-trained Q-learning bot as the second place.

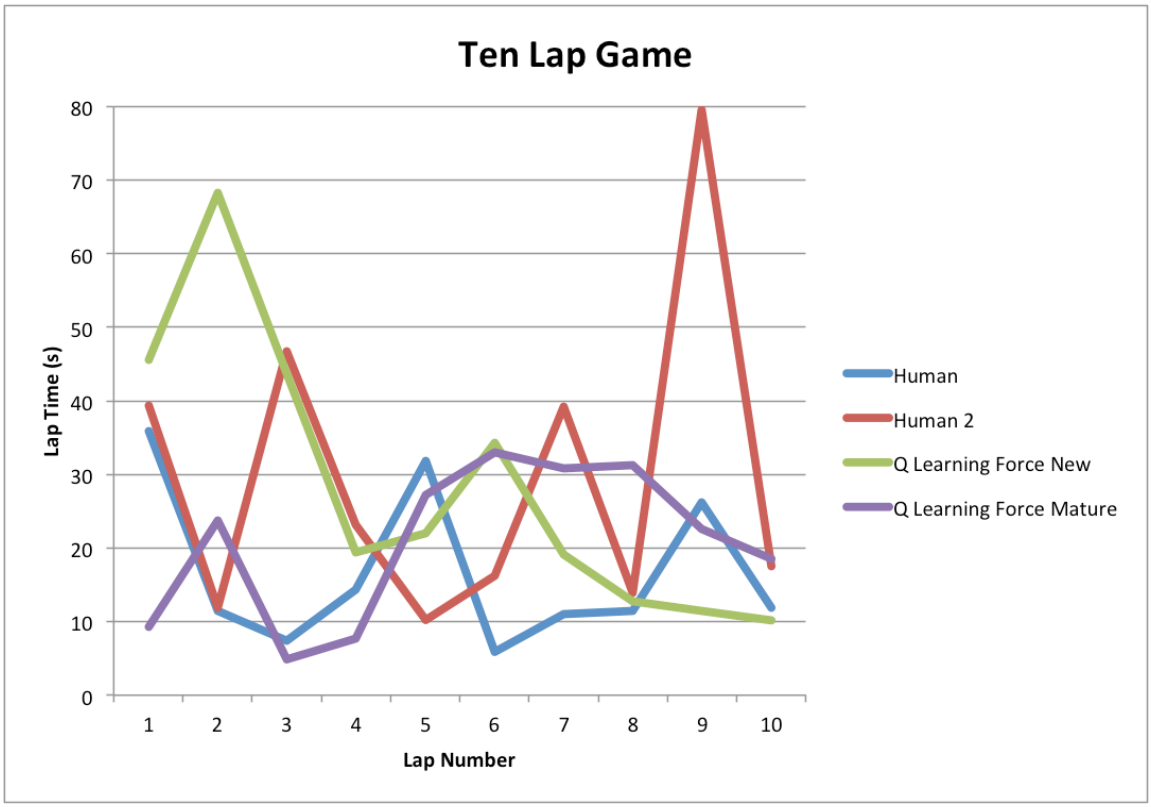


Figure 11: Two Humans and Two Q-learning AI Lap Times

What We Learned

Random Agent

The first thing that became apparent when we began this project is the importance of using good programming practices. The original team that created Magnet Racers had not been adept at Object Oriented Programming yet. This led to cluttered and difficult to read code. The first major portion of our assignment became cleaning the game up so that implementing AI agents would be more manageable. After the code was cleaned up, the team was able to create the random move AI agent without too much difficulty.

Decision Tree Agent

From developing the Decision Tree AI Agent we learned the importance of play testing our game and being able to format basic tactics to win. Luckily for us, there is only one decision the bot needs to be able to make at any time: do I toggle my polarity or not? We understood from play testing that we wanted to have our polarity match the pole behind us and push the magnet through the track. This is what led us to creating the initial iteration of our DT Agent which sectioned off the game into quadrants. While the agent was able to complete the race it would run into issues such as bouncing in a corner where it took so long to get out of the spot,

the AI was no longer competitive to a human racer. This led us to digging deeper to find out how to add more checks to avoid pitfalls such as that as well as develop additional tactics to become more competitive to users. We are happy with the final iteration of our DT Agent and believe it can be competitive against new players, however with more time and research we realize that there is a lot more complexity that can be added to make this particular agent a much more competitive racer. And that due to the chaotic nature of the game, it would be extremely difficult to make an AI agent be able to match a human player in strength.

Q-learning

After this we actually took our step into working on the AI. Magnet Racers is a very skill based game, that demands experience from the players. The players with greater experience playing the game usually develop better skill at playing the game. This made it difficult platform to start building our AI, since giving the AI ability to play the game required us to get in depth knowledge of how and what the best strategies are to play such a nuanced game. The game is very simple with just one input but has very subtle tactics that result in better performance in game. These tactics are based in the physics of the game. Bringing into perspective that designing an AI has specificity to it that change from game to game. No 2 game AI development projects could be carried out with similar processes in mind. You always have to move on with techniques that are unique to the game itself.

We were able to develop AI agents with varying levels of complexity, but what all of these agents had in common was how the aspect of player experience contrasted every AI agent in and out of development. We talked about how player experience plays a big part in our game earlier, even still this aspect isn't limited to game projects such as ours. It is sure to be extended in any game AI project just in varying effects, since player experience was always a big deal for our project it was something that we were able to judge for ourselves and learn from it new ways of looking at game AI.

These challenges became very apparent while implementing the Q-learning agent. Our agent had to attempt to learn to cope with the forces created by not only fixed poles, but also moving opponents. Furthermore, our agent had to learn effective moves even while it was subject to having momentum from previous actions. Do to the challenges facing the agent, the team had to develop multiple different reward systems before meeting with success. Additionally, the team had to tune the different parameters involved in the Q-learning algorithm in order to create a proficient agent (learning rate, discount factor, probability to explore).

In conclusion, the team had to gain an in depth knowledge of the game in order to be able to create AI agents that could play the game. The team also had to gain an understanding of each of the algorithms implemented in order to make them work effectively. Finally, the team gained more experience with Unity and got a better idea of the effort involved in making adept AI characters.

Appendices

Appendix A: Learned Q Map with Force Rewards

(Limited content to 1 page for brevity)

40
-1;3;1;3;-1;0;22
1;3;1;3;-1;11;0
1;3;1;0;-1;0;19
-1;3;1;0;-1;0;22
-1;3;1;0;1;0;22
-1;3;1;1;1;0;22
-1;3;1;2;1;15;0
-1;3;1;3;1;0;4
1;3;1;3;1;0;0
1;3;2;0;-1;22;0
-1;3;2;0;-1;0;22
-1;2;2;0;-1;22;0
-1;2;2;0;1;22;2
-1;2;2;1;1;19;0
-1;2;2;1;-1;4;0
1;2;2;1;-1;0;21
1;2;2;1;1;0;21
-1;2;1;1;1;0;0
-1;2;1;1;-1;0;20
-1;2;1;0;-1;0;2
1;2;2;0;1;0;22
1;2;2;0;-1;0;22
1;2;2;3;1;0;21
-1;2;2;3;1;17;0
-1;2;2;3;-1;6;0
-1;2;2;2;1;8;0
-1;2;2;2;-1;0;0

Appendix B: Example of Timed Lap output for Q Learning Agent

(This output is for an agent which was rewarded based on forces acting on the agent after an action was taken)

Lap : 1 Time: 63.431718
Lap : 2 Time: 30.361594
Lap : 3 Time: 20.435401
Lap : 4 Time: 13.766895
Lap : 5 Time: 22.846006
Lap : 6 Time: 13.434323
Lap : 7 Time: 20.883733
Lap : 8 Time: 7.431839
Lap : 9 Time: 34.103113
Lap : 10 Time: 12.968917
Lap : 11 Time: 35.382928
Lap : 12 Time: 13.367414
Lap : 13 Time: 35.10066
Lap : 14 Time: 7.531057
Lap : 15 Time: 30.377491
Lap : 16 Time: 8.247848
Lap : 17 Time: 13.799346
Lap : 18 Time: 5.221377
Lap : 19 Time: 16.294564
Lap : 20 Time: 7.89691
Lap : 21 Time: 23.844044
Lap : 22 Time: 6.135547
Lap : 23 Time: 33.020942
Lap : 24 Time: 4.822972
Lap : 25 Time: 8.512678
Lap : 26 Time: 9.343684
Lap : 27 Time: 18.007368
Lap : 28 Time: 4.373033
Lap : 29 Time: 14.299346
Lap : 30 Time: 3.774067
Lap : 31 Time: 51.494974
Lap : 32 Time: 5.752657
Lap : 33 Time: 10.858007
Lap : 34 Time: 10.790796
Lap : 35 Time: 4.888004
Lap : 36 Time: 6.618067
Lap : 37 Time: 19.954305

Lap : 38 Time: 6.648628
Lap : 39 Time: 9.411031
Lap : 40 Time: 36.49709