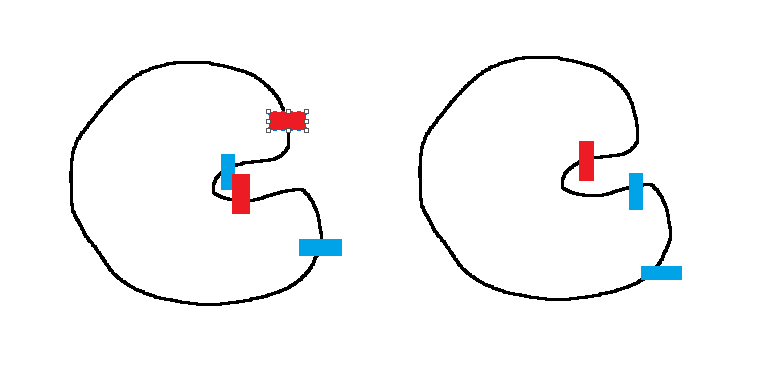
Order of paper – What is to come

|  |  |
| --- | --- |
| Controls - 291 | |
| Initial Design | The initial design of the gameplay would consist of the player driving around the track as fast as possible. The player would fully control the car with on-screen buttons: Accelerate, Brake, Reverse and Turning. While turning and braking, the player could drift the car slightly to allow them to take sharper corners at higher speeds. Mastering of the drift mechanic will allow players to achieve faster lap times. |
| Implementation | Due to the mobile nature of the game, most of the game can only be operated with 2 inputs (thumbs). With playtesting, I found that players struggled to control all the movements needed, as they would have to take their fingers off the accelerator to turn. To remedy this, I made the car auto-accelerate as well as combine the brake and reverse function, so the brake button caused negative acceleration, rather than reducing the speed to 0.  Originally, drifting would start as soon as the car was braking and turning. This proved to be troublesome when there were slight turns in a track. To remedy this, I created a check on the sideways velocity of the car. Once it was above a margin, the drifting system would start. |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well | The simplified car mechanics allowed play-testers to manoeuvre most tracks with ease, I did receive some feedback that said allowing the player to have full control of the car could have some benefits on more complex tracks, but I found in most cases the car’s mechanics were suitable.  The drifting system was not perfect as it required the player to brake to use it. The brake button returned a Boolean (0/1), while I would have liked the drifting to be more intricate allowing the player to change the amount of braking (0.1 ->0.5 -> 1.0). |

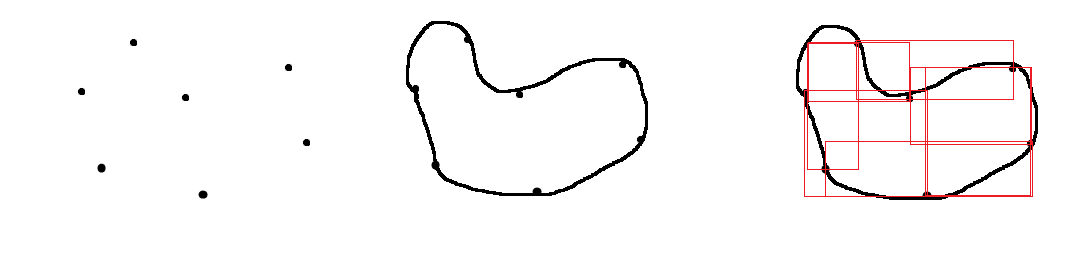
|  |  |
| --- | --- |
| Checkpoints - 391 | |
| Initial Design | To ensure the player is going around the track the right way, I wanted to add a checkpoint system to each track. These would have functions attached to them which were used by the race manager to check where the player was and to allow the player to return to them if they drive too far off the track. |
| Implementation | Each checkpoint is spawned with a positional ID, finish line bool and a collision trigger bounding box. When the player passes through the trigger, the checkpoint system checks to see if the ID of this checkpoint is next after the last passed through checkpoint. If it is not, then the player is directed towards the checkpoint they need to pass to progress. If the checkpoint passed though is the finish line, then a new lap is started and the last passed checkpoint is reset.  Due to the nature of the track’s generation, the checkpoints would have to be part of the creation process. Initially, I had the checkpoints made where each of the points of the track were. However, I found that the checkpoints could get too close together and that would interfere with the collision check required for the checkpoint system. I instead created a system which would create points along the path of the track that were equidistant.  To allow for different styles of play, the bounding box for the checkpoints would have to span the width of the track, I needed to spawn the checkpoints, perpendicular to the track. However, I could not find a mathematical way that produced the results I needed so instead I made the checkpoints look at their second control point on the curve.  The time it takes to go between checkpoints is recorded and used when analysing the players behaviour. The total time of each time between checkpoints is aggregated to produce the lap time and then the lap times are aggregated to produce the final race time. |
| Benefits / Pitfalls | I made the checkpoint system before I added segments to the track design, but if I were to redesign the system, I would move the checks to the segments as that would remove the need for the positioning and rotation processes. |
| Worked Well / Didn’t Work Well | Initially I wished the checkpoints to be invisible, but with testing, I found that players found it difficult to know where they had to pass. Subsequently, I made all the checkpoints visible.  If I could have made a way for the points to be perpendicular, I would have implemented it, the Look At method did not always produce the best results, but they were always within my margin of error. |
| Figure | Checkpoint Visual  Checkpoint System. Before/After  Pseudo code of checkpoint system. |





Path Generator

|  |  |
| --- | --- |
| Track | |
| Initial Design | Tracks are stored as a sequence of 2D points, the path generator uses these points to create a closed Bezier curve. By performing tests on the points and path, most of the aspects of the track can be found such as Length, Height, Width, and the number of straights and corners. |
| Implementation | Originally, I had planned to stop here when it came to represent the track, but I started to develop the evolutionary algorithm I found that I needed a more in-direct way for it to view track as. In my research, I found a paper that evolved tracks by cutting the track into segments and then mutating some segments during the evolution process. I decided to use this methodology for my tracks, so I then added a process that creates segments from a track.  Each segment is comprised of 3 points, the start, middle and end points. Using vector maths, the distances and angles within the segment are calculated. With these measurements, it can be determined what direction the segment turns as well as its size and area. This is done for each point in the track to create an array of segments, which can be used in other processes. |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well |  |
| Figures | Point Track  Path Track  Segment Track |



|  |  |
| --- | --- |
| Random Track Generation | |
| Initial Design | Random tracks were required so that I could test the algorithm tracks vs a random track to test if there was a noticeable difference. My requirements for a random generated track were:  Diverse – If every random track looks the same, it allows the player to make the distinction.  Closed Circuit – Allows for multiple laps  Feasible – The player must be able to finish the track |
| Implementation | I originally tested the straightforward methodology introduced by Togelius (2007) but found that tracks were still to similar. To try to fix this I decided to use elements of the methodology for the initial generation by making the original shape with a convex hull instead of a rectangle.  The convex hull was generated by creating a collection of randomly placed points and calculating the smallest polygon that encloses all the points. Instead of modifying the existing points, I decided to add points in-between the current points. These new points were randomly placed within the bounds of their parent points. |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well |  |
| Figure | Straightforward / Radial / Random Walk  Straightforward vs Convex Hull  Midpoints |

|  |  |
| --- | --- |
| Track Checks | |
| Initial Design | As I built up the track system, I required more data from a track to see if it is feasible to drive on. Checks for intersections and closeness were added. Initially I believed that allowing a track overlap was a problem, so I designed a way to check if certain lines overlapped. It would take the straight lines between each sequential point and pass them through a geometric equation that checks if the lines intersect.  To determine the difficulty of the track, the variance in segment angles is analysed. Shallow angles are considered easier than sharper angles, but continuous sharp angles or changes of direction also factor into the difficulty of the track. |
| Implementation | As I had originally made the overlap check only consider linear lines between points rather than the path it would take, it led to tracks with intersections pass the check. I changed the formula to consider the path instead and there was a significant decrease in overlaps.  However, there would still be tracks that had intersections, especially from tracks made by the algorithm. As I could not find a way to eliminate them, I decided to instead check the angle of intersection. If an angle was too small (>20°), it would fail the check. This proved to be the most effective method because it also led to more track diversity. |
| Benefits / Pitfalls | The only error I was not able to fix was when a corner was very tight and overlapped. Because the intersection angle would be within the bounds, it was not flagged but the corner would be hard to take for player. It happened infrequently enough to not be a large problem. But given more time I would have liked to find a method to fix it. |
| Worked Well / Didn’t Work Well |  |
| Figures | Difficulty Equation  Severe Angles  Error Example |

|  |  |
| --- | --- |
| Player Tracking | |
| Initial Design | Each player can design their own profile with their own username and icon, this is not linked to the internal player profile used for tracking data. That profile is only created once they have read and passed the consent pages. For example, if someone below the age of 18 enters their age they will be able to access the game, but an account will not be made for them. I decided to let users under the participation age to play the base game with no data-tracking, so no player-based tracks, so that they are not encouraged to lie about their age in order to play the game. |
| Implementation | In the first concepts for the game, I wanted to track various telemetry from the player such as play time, exit events, and app navigation. However, I had not taken into consideration what data I could gain from these metrics. Eventually, I found I instead needed to outline what data I needed for the algorithm and then figure out what metrics could provide that data.  Likes and Dislikes – Did the player enjoy the track?  Points of Enjoyment – What did the player like about the track?  Points of Hindrance – What did the player dislike about the track?  **Track Selection**  I assumed that tracks that were picked could be linked to how much they enjoyed that track. I believed that players repeatably driving on the same track rather than new or different tracks showed that they were getting something from the track, be it enjoyment or pursuit of improvement. When a player displays these qualities, they should be catered for.  The history of tracks is also recorded and used in the evaluation process. Tracks played more recently have a larger impact on what a new track is tested against.  **Improvement / Problem Areas**  By tracking how long it takes the player to go through a segment, an average time and speed is created for similar segments. By tracking the difference from the average, it shows where the player is improving or encountering problems. These are then fed into the ideal track for new tracks to be tested against.  When it comes to measuring player data, it can be very difficult as inherently human behaviour is qualitative information. Being able to quantify that data is  Instead of using an indirect method of assuming what data corresponds to metrics I decided to directly ask the player for this data. I believed that I would not be able to produce a system that would be able to accurately infer the player data, which could affect the evolution process.  **Time**  The local time since the track was last played is also stored. This is done to try to lessen their impact on the genetic algorithm if they have not been played recently. |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well | If I had been able to, I would have liked to create an indirect method of modelling a player. |
| Figures | Time Falloff |

|  |  |
| --- | --- |
| Ratings and Requests | |
| Initial Design | Once the player has finished a track they are presented with a ratings and request screen. They can then like or dislike the track and request an increase, decrease or no discernible change in the categories stored by the ideal track.  Initially I wanted all the requests to be available for the player to choose from, this would allow for better optimisation of the ideal track. However, because of the limited screen space available on mobile devices, I decided to limit it to 2 requests.  Once the requests are completed, they are applied to the ideal track.  I also included a text box which let the player give any additional feedback during testing. While I like to think I had covered most of the criteria, having actual feedback is invaluable especially for ideas I had not thought about. |
| Implementation |  |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well | I would have liked to make a way for the requests from every played track matter rather than only the most recent tracks as this would have allowed for better optimisation of the new tracks. |
| Figure | Basic Request Screen |

HOW THE GENTIC ALGORITHM WORKS

|  |  |
| --- | --- |
| Genotype/Phenotype Representation (TrackDNA) | |
| Initial Design | Genotype  After each race, a chromosome (Track DNA) is created, it contains a combination of the standard track data (points, segments, measurements) and its corresponding player data. Different systems in the algorithm use part of the chromosome for their processes. For example, the crossover and mutation systems only require the segment data. |
| Implementation |  |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well |  |
| Figures | TrackDNA |

|  |  |
| --- | --- |
| Initial Population | |
| Initial Design | When the player first starts the game, they are presented with a selection of 3 human-made and 3 random tracks. Once a player has successfully played a track and given a rating, it is then added to the initial population. To create a new track, the initial population needs to be larger than 8, after it has passed that threshold the player is given the option to generate a new track. The algorithm then uses this initial population in |
| Implementation | The human tracks and random tracks are indistinguishable as not to give bias to certain tracks. They are also sorted into a random order when the player first starts the game as another method to attempt to avoid bias [ANCHORING]. |
| Benefits / Pitfalls | I would have liked to have tested if the initial population size affected how the generated tracks performed. By requiring a smaller generation, the player would have access to a new track much sooner, but it may not be as optimised as tracks generated from a larger population base.  When it comes to mobile gaming, being able to keep first-time players is one of the most important requirements for a successful game. So, being able to access to core of the game quicker could lead to a more popular game.  I decided on 8 tracks as it ensures a track has been played multiple times, hopefully multiple tracks. The player is not shown the exact number they need to complete, as I was planning on testing different sizes. |
| Worked Well / Didn’t Work Well | -scaled up |
| Figure | UI for showing Progress towards track |

|  |  |
| --- | --- |
| Fitness Function | |
| Initial Design | When a track is first generated it must first pass the check to ensure it is a feasible track, any track that fails is given a low score. The new tracks are then tested against an **ideal track**, which is created from a combination of player requests and gameplay. It stores the supposed best values in the following categories: Length, Straight Count, Corner Count, Speed, and Difficulty.  It also stores things to include and avoid, such a straight where the player got a fast speed or a corner that has given the player problems. Tracks that contain any problem areas are rated much lower.  The fitness is stored as an array of numbers, each with different weights. Each element of the array corresponds to a category within the ideal track. The elements can go from 0, which is a fail, to 1, which is considered perfect.  Direct inputs from the player are weighted higher than inferred values. The weighted sum is then output as the final fitness. |
| Implementation |  |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well | This is the part of the project I wanted to improve more. It relied too much on assumptions to be effective.  Currently, the weightings are fixed, with further development they could vary to help better represent the player. |
| Figures | Ideal Track  Vector Fitness 8 elemtns  Vector Scale 0 - 1  Weightings |

|  |  |
| --- | --- |
| Parent Selection | |
| Initial Design | Once the initial population has been filled, the crossover parent selection is based on the probability which is proportional to their fitness. The method for selection I used was Stochastic Universal Sampling (Baker 1987), which uses 2 fixed points to output both parent within the same process. This methodology worked well, as it meant better tracks had a larger impact on the child population. |
| Implementation |  |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well |  |
| Figures | Stochastic Universal Sampling |

|  |  |
| --- | --- |
| Crossover | |
| Initial Design |  |
| Implementation | When the 2 parent tracks need to be compared, a weighted random number based on their fitness is used.  To avoid as many issues as possible the 2 parents are equated. They are compared to find out the desired point count and rotation. The count is a random integer within the bounds of the parent tracks point total. If they both have the same rotation, there is no change but if they are opposite then it is chosen from the weighted number.  The 2 tracks are scaled to the new point count with the same script used to create checkpoints. The tracks are then centred on 0,0 and changed to have the same rotation.  Then the corresponding segment types on each track are compared and stored. If the corresponding segments have the same direction, then the child track will have the same direction at that point. If they are opposed, it is based off the weighted number. This process also had a chance to mutate, receiving a random segment direction instead.  The child track then has a list of desired directions and would iterate through each direction and find a related segment from the parents to add. The result would be a list of segments which are then placed from the origin. After this the track is centred and rotation changed if needed. |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well | The parent scaling would sometimes change the track too much as the checkpoint system used would only move the main points and not consider the other control points. This would change the dynamics of the track, but I found no realistic way to fix it.  I wanted to implement a further way for the child track to analyse the segments for their height and width, as currently it would only consider the angle and that could lead to unwanted overlaps and bizarre tracks.  I also would have liked to have different options for mutations, as presently it only changes the segment type, which still uses the parent segments. Being able to generate a segment with new values could have helped keep tracks diverse. |
| Figures | Parent Scaling    Child Making 😉 |

|  |  |
| --- | --- |
| Survivor Selection | |
| Initial Design | Once there have been enough child tracks created, they are ranked based on their fitness. Initially, 50 tracks were generated with the top 10 considered for selection. This was done as an attempt to ensure enough viable track were made. One of the top 10 was then selected with a weighted random number based on their fitness. |
| Implementation |  |
| Benefits / Pitfalls |  |
| Worked Well / Didn’t Work Well | As I never got to directly test the algorithm with players, I could not make changes based on the data I would have received. If it was possible, I would have tested the evaluation process and how population size has an impact on track enjoyment. |

The player would then be able to play this track and the process would repeat. The

Future

* Custom Cars -Based on Playstyle