Title –

Name – Isaac Fagg

Student Number – S4920008

Abstract –

I wish to request that my dissertation is assessed in accordance with marking option:

1: Lit Review 1400 words (30%); Dev Report 3000 words (55%)

[Introduction 4](#_Toc41503243)

[Background/Context 4](#_Toc41503244)

[Research 4](#_Toc41503245)

[Procedural Generation 4](#_Toc41503246)

[Search Based Procedural Generation 4](#_Toc41503247)

[Adaptive Games 5](#_Toc41503248)

[Personalised Content 6](#_Toc41503249)

[Personalised Tracks 6](#_Toc41503250)

[Random Track Generation 7](#_Toc41503251)

[Development & Implementation Report 8](#_Toc41503252)

[Gameplay 8](#_Toc41503253)

[Checkpoints 9](#_Toc41503254)

[Track 10](#_Toc41503255)

[Random Track Generation 11](#_Toc41503256)

[Track Evaluation 12](#_Toc41503257)

[Player Tracking 13](#_Toc41503258)

[Ratings and Requests 14](#_Toc41503259)

[Evolutionary Algorithm 15](#_Toc41503260)

[Genotype Representation 15](#_Toc41503261)

[Initial Population 15](#_Toc41503262)

[Fitness Function 16](#_Toc41503263)

[Parent Selection 18](#_Toc41503264)

[Crossover 18](#_Toc41503265)

[Survivor Selection 19](#_Toc41503266)

[Self-Assessment of Learning 19](#_Toc41503267)

[References 19](#_Toc41503268)

[Figure 1 - Initial World Generation. All chunks are assigned their value at initialisation. 4](#_Toc41503269)

[Figure 2 - Continuous World Generation. A chunk is not defined until it first needs to be generated. 4](#_Toc41503270)

[Figure 3 - Criteria scores for all games. (Browne 2008) 5](#_Toc41503271)

[Figure 4 - Popular search-based procedural generation methods and the new Adaptive Model. (Liapis 2012) 6](#_Toc41503272)

[Figure 5 - Random Walk (a) and Radial tracks (b) (Togelius 2007) 7](#_Toc41503273)

[Figure 6 - Population score using Ranking and Like/Dislike. (Cardamone et al. 2011) 8](#_Toc41503274)

[Figure 7 - Gameplay 9](#_Toc41503275)

[Figure 8 - Checkpoints generated too close as well as wrong rotation. 10](#_Toc41503276)

[Figure 9 - Point to Path representation 10](#_Toc41503277)

[Figure 10 - A segment with measurements 11](#_Toc41503278)

[Figure 11 - Segment Bounds showing each point is part of 3 segments. 11](#_Toc41503279)

[Figure 12 - Convex Hull generated from randomly placed points 12](#_Toc41503280)

[Figure 13 - Midpoint created within parents bounds. 12](#_Toc41503281)

[Figure 14 - Example of tight corner 13](#_Toc41503282)

[Figure 15 - Time fitness graph 14](#_Toc41503283)

[Figure 16 - Post race request prototype 15](#_Toc41503284)

[Figure 17 – Track DNA 15](#_Toc41503285)

[Figure 18 - Track progression. 16](#_Toc41503286)

[Figure 19 - Ideal Track Code 17](#_Toc41503287)

[Figure 20 - Fitness vector 17](#_Toc41503288)

[Figure 21 - Weightings of the fitness functions. The values were never tested. 18](#_Toc41503289)

[Figure 22 - Stochastic Universal Sampling 18](#_Toc41503290)

[Figure 23 - The same track before and after scaling. 19](#_Toc41503291)

# Introduction

# Background/Context

Procedural generation (PG) has had a place in the game industry for more than 40 years. Currently PG is used in a limited capacity in games used to help aid development by lessening work on certain aspects of a game. Games like Minecraft and No Man’s Sky use random seed PG instead of human-created content to create infinite worlds. However, these worlds are built using the same standard rules meaning a biome or planet in one world offers little difference from the same biome in a different world. If instead the initial world creation (Figure 1) or continuous generation (Figure 2) was based on data gathered from the player, it could lead to interesting and different player experiences by offering new personalised biomes.

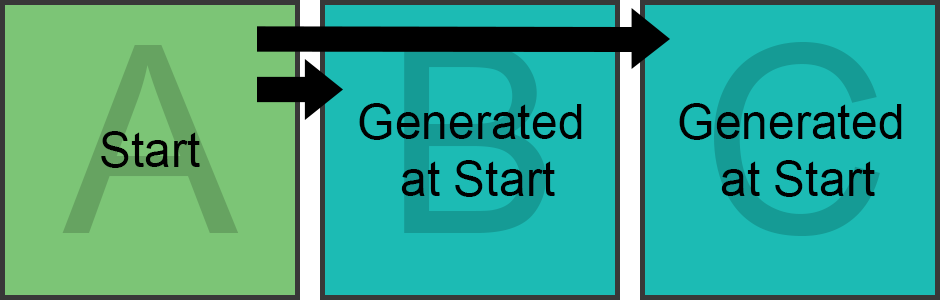


Figure - Initial World Generation. All chunks are assigned their value at initialisation.

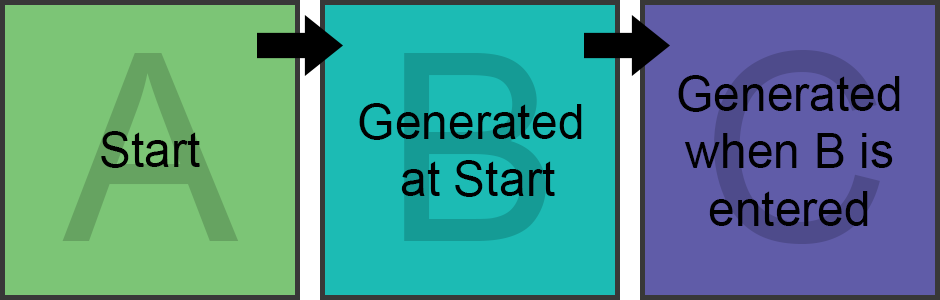


Figure - Continuous World Generation. A chunk is not defined until it first needs to be generated.

# Research

## Procedural Generation

Sorenson and Pasquier (2010) a

## Search Based Procedural Generation

Search-based procedural generation (SBPG) is a type of PG that relies on generating and testing content that fits certain criteria (Togelius et al. 2010, 2011). The tests performed do not just pass or fail content, instead they are assigned a fitness. A standard PG algorithm will only construct a single instance based on rules set by the designer. With SBPG multiple instances are created and then compared to attempt to generate the best content. While not essential, it is commonly used in conjunction with an evolutionary algorithm to perform the task of creating better content.

There are various approaches to SBPG, but the most popular are Predefined Evaluation, which has a set of established rules from which the content is tested against, and Interactive Evolution (Takagi 2001) where the user is the source of the testing by assigning a fitness based on the user’s selection or rating.

An example of Predefined Evaluation is a study by Browne (2008) which used SBPG to design rulesets for board games where the fitness based on the how the game performed. Some games did not use all the criteria available depending on the rulesets they had, all the criteria are shown in Figure 1. They found that measurements made during the evolutionary process were unreliable but proved useful for establishing if a game was viable.

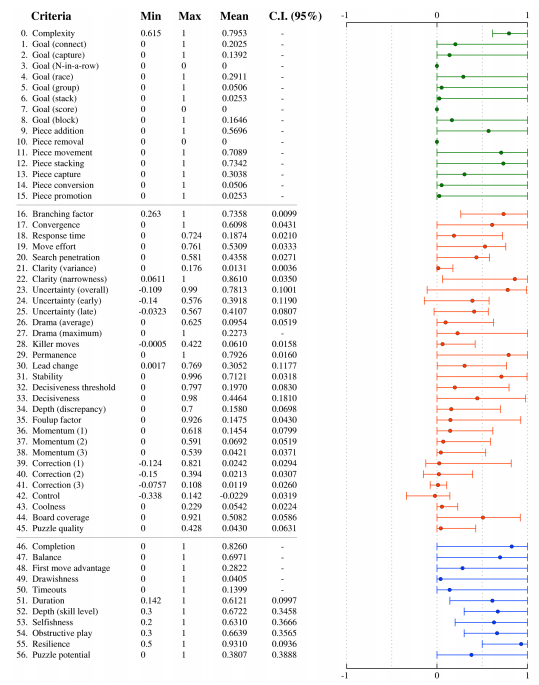


Figure - Criteria scores for all games. (Browne 2008)

A different study by Hastings in 2009 used Interactive Evolution SBPG to optimise weapons in a space shooter. Weapons were given a fitness based on how often they were used compared to the total time the player had access to them, with the most popular being used to create new weapons.

While these methods are popular, they are ill-equipped for personalised content. Using predefined fitness values does not allow much flexibility in what content is created but can be useful in the cases of checking if content is viable. Interactive Evaluation suffers as content is only assigned a fitness when it is selected, which can lead to the content becoming similar.

## Adaptive Games

Liapis et al. (2012) explored a new method of search-based procedural content generation, which changes the fitness function based on the player as shown in Figure 3. By selection content the entire fitness function is changed, so past content within the population is judged differently. This is done with the hope that as the player develops, their preferences do as well. The paper focused on presenting the potential of the method rather than proving its advantage over the other methods.

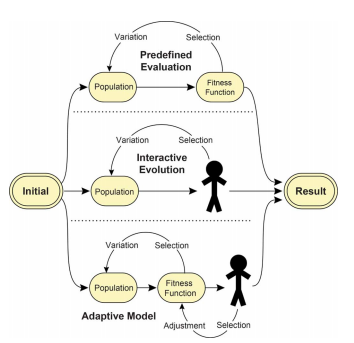


Figure 4 - Popular search-based procedural generation methods and the new Adaptive Model. (Liapis 2012)

## Personalised Content

The games industry has seen increased interest in automated personalised content as it would allow games to cater to a wider audience. There are types of personalised content already prevalent in games such dynamic difficulty (Hunicke 2005) which changes the difficulty of the game based on how the players performing.

However, personalised content is reliant on how the user’s data is interpreted by the system. Karpinskyj et al. (2014) categorises how players differ into 5 sections:

1. Preferences (What parts of the gameplay is engaging)
2. Personality (Character of player)
3. Experience (How players respond while playing)
4. Performance (Rate of player progression)
5. In-Game Behaviour (Player actions)

The paper goes on to survey games within these categories and found that personalisation requires meaningful characteristic selection, observation of that characteristic that is reliable and practical, integration of the observational method into the personalisation system, and evaluation that proves the personalisation has performed appropriately. Many of the games surveyed failed to properly shows that the personalisation had the intended impact.

Pedersen et al. (2010)

## Personalised Tracks

Papers by Togelius et al. (2006, 2007) attempted to make racing fun by using player modelling to assist track evolution. Whilst there was no previous research on what made racing fun, they used their own experiences to select what they believed racing to be about:

* High Speed – People like to drive fast, so tracks should allow for a high maximum speed.
* Challenging to drive – Driving on a straight track is not interesting.
* Constantly crashing is not fun – Try to avoid locations where the player could crash.
* Variety in challenge – Repetitiveness should be avoided.
* Drifting and skidding – Pushing the limits to get the lowest time.

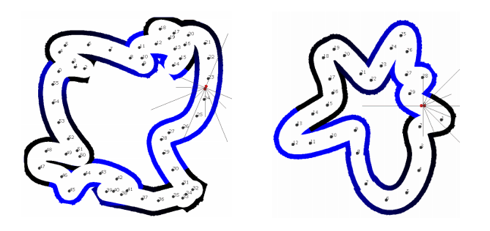
They introduce their evolutionary algorithm called Cascading Elitism, which was used both for their tracks and the AI agent they taught to drive like the players. This system starts with a large population (100 in this example) and multiple fitness criteria. It starts by testing them based on a fitness criterion. Only the best 50% are tested on the next criterion and so on for each criterion until 15% remains, which then fill the population back to 100 with mutated copies. However, they did not use crossover the new population so they could eliminate risks of competing conventions.

When considering if a player model is appropriate, they state that it is difficult to gather the data from just a few laps of the track, instead they taught the agent to imitate the player to allow them to create more data.

Though they were able to create agents that closely resembled the performance of human drivers (total lap times), they were unable to have it effectively replicate actual human behaviour. They state this may have been due to the reactive nature of their tracking as well as the limited inputs they gave it.

## Random Track Generation

In order to create an initial population random tracks were needed. Togelius et al. experimented with different methods for random track generation: Straightforward, Random Walk and Radial. Straightforward tracks are generated by creating a rectangular shape and then using Gaussian mutation to perturb the points coordinates. Random walk starts the same way but iterates through each point and mutates it. Radial tracks places equally spaced points around the centre at a random distance. They found that Random walk produced tracks that could be too difficult to drive on and Radial method tracks made tracks that all looked very similar.



1. (b)

Figure 5 - Random Walk (a) and Radial tracks (b) (Togelius 2007)

A study by Cardamone et al. (2011) presented a framework for using Interactive Evolution for track design. Players would rate tracks after playing them and these tracks would be used to evolve future tracks. The population was shared with other users, so tracks produced were not tailored for the player, but it was shown to produce interesting and quality tracks. One of the tests they performed was how the rating system differed results. The player could either rank from 1 to 5 or like and dislike a track. Participants states they preferred the like/dislike interface as they felt they could express their preferences better, as they found they could not provide meaningful rankings with the rating system.

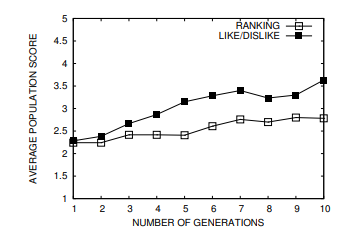


Figure - Population score using Ranking and Like/Dislike. (Cardamone et al. 2011)

Cardamone et al. (2014) a

# Development & Implementation Report

Paper Order

## Gameplay

The initial design of the gameplay would consist of the player driving around the track as fast as possible. The player would complete 3 laps by controlling 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.

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.

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).

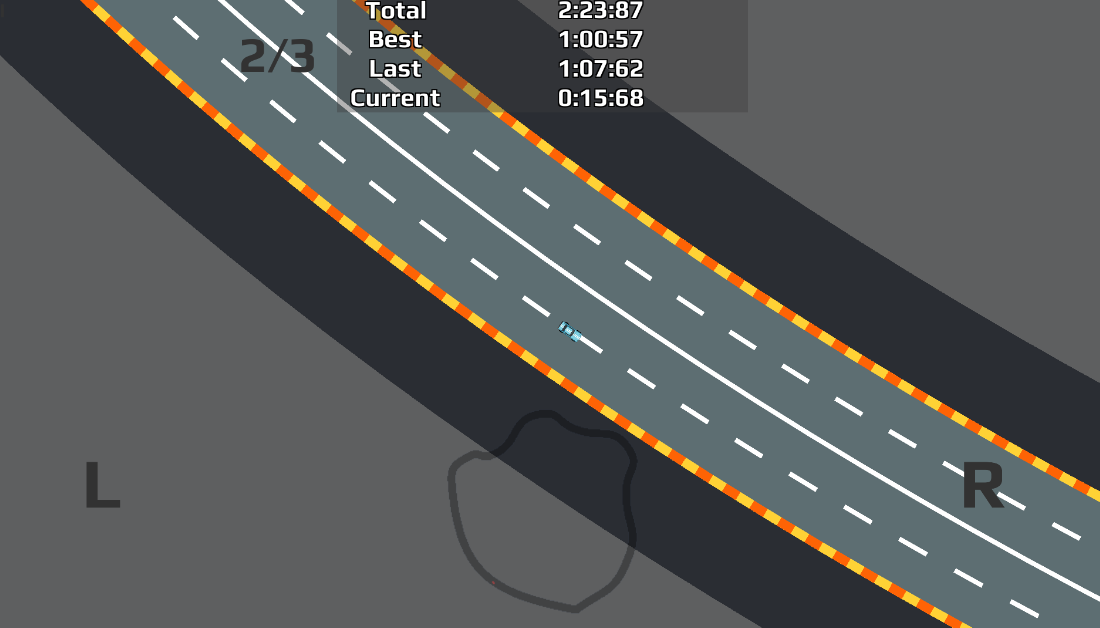


Figure - Gameplay

## Checkpoints

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.

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.

Initially, I had the checkpoints generated on the track control points. However, I found that the checkpoints could get too close together and that would interfere with the collision check required for the checkpoint system. To fix this I created a system which would create points along the path of the track that were equidistant.

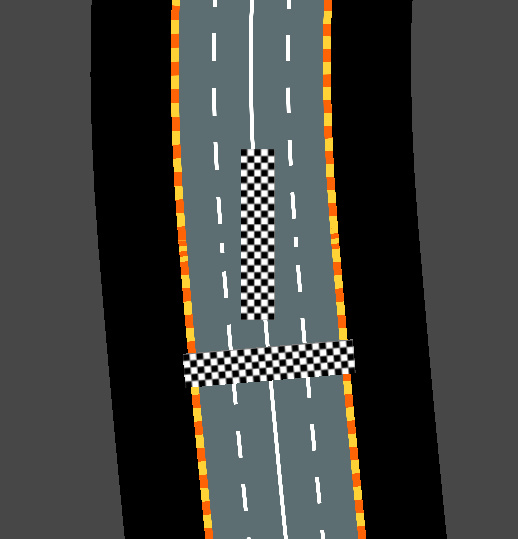


Figure - Checkpoints generated too close as well as wrong rotation.

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.

I made the checkpoint system before I added segments to the track design, but if I were to redesign the system, I would move to check the players progress through a segment.

## Track

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.

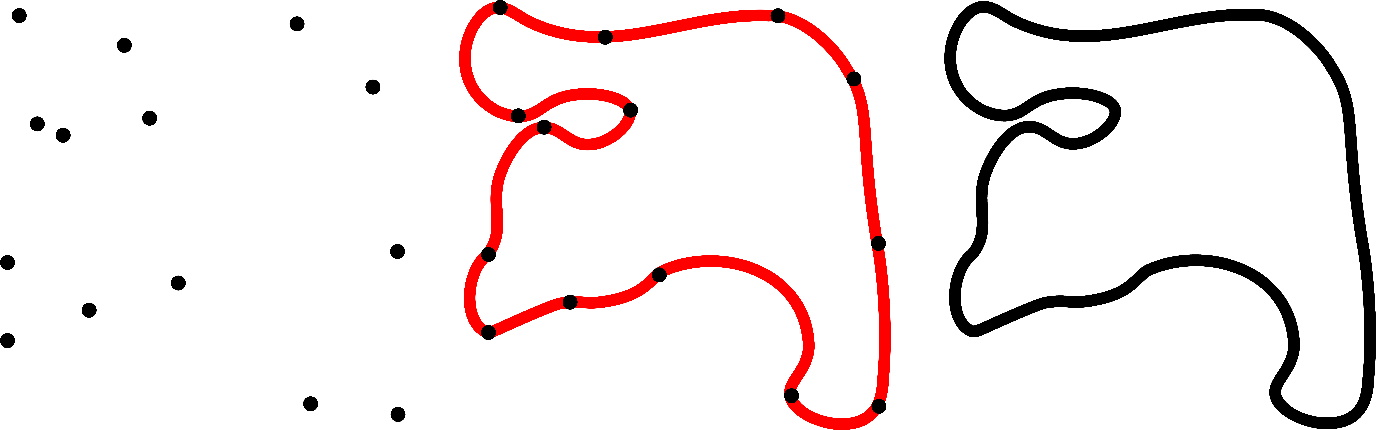


Figure - Point to Path representation

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. I decided to adopt the methodology presented by Cardamone et al. (2011) which splits the track into segments.



Figure - A segment with measurements

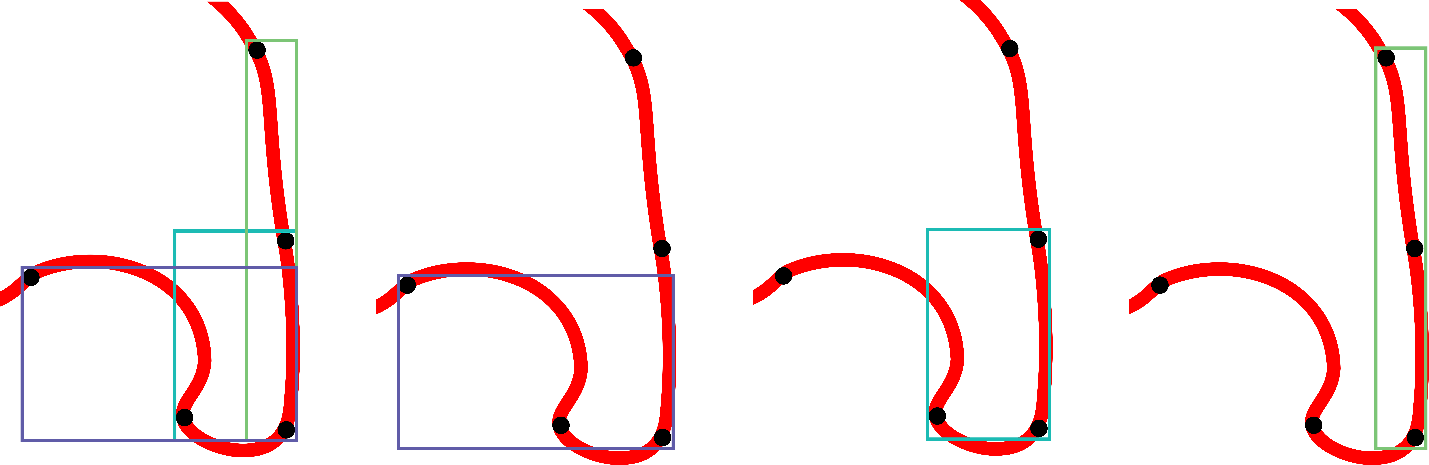


Figure - Segment Bounds showing each point is part of 3 segments.

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.

### Random Track Generation

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

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 (Figure 10). 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 (Figure 11).

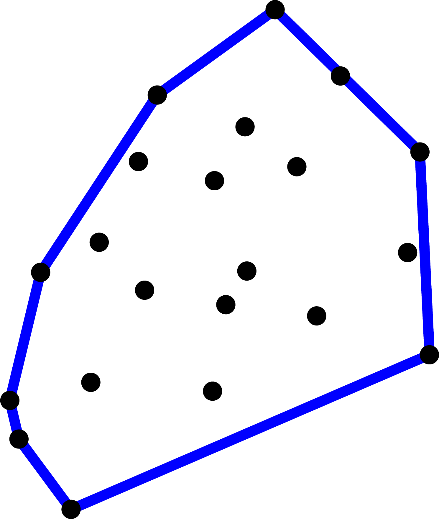


Figure - Convex Hull generated from randomly placed points

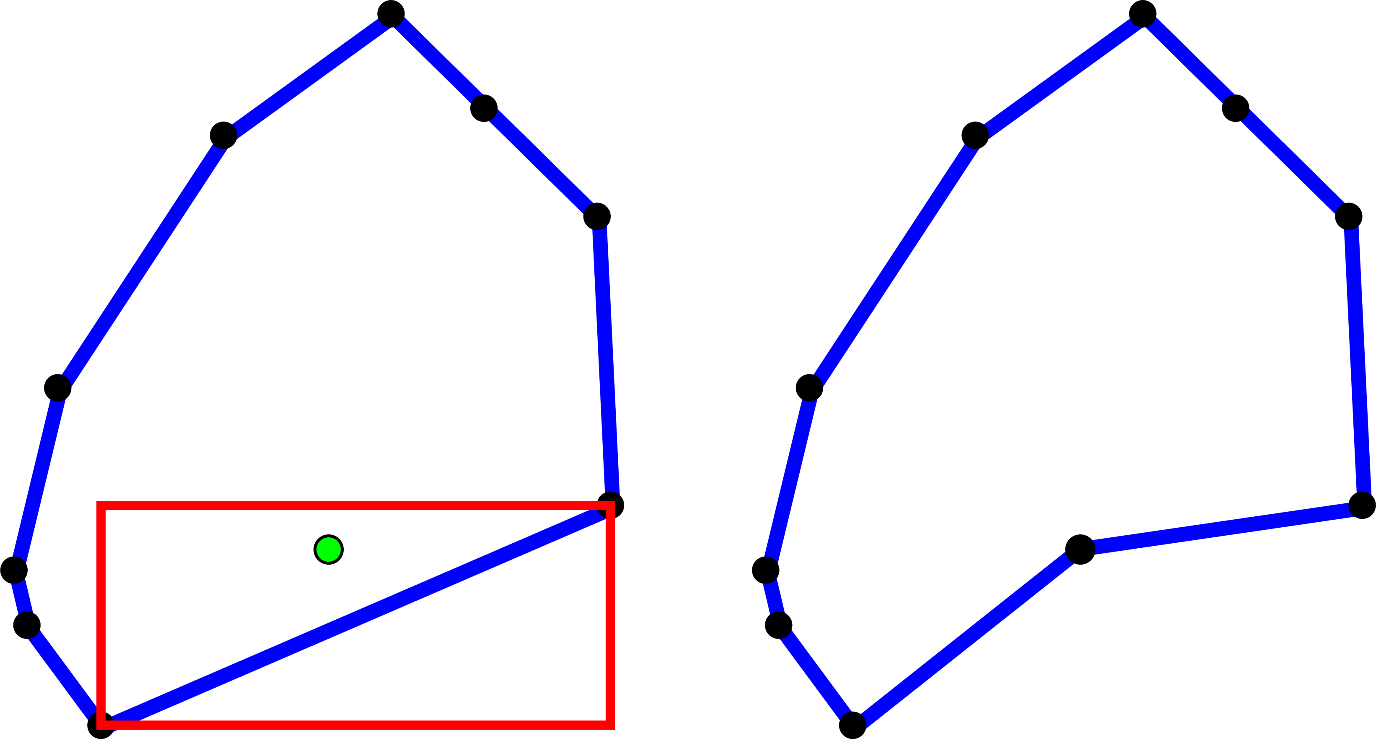


Figure - Midpoint created within parents bounds.

## Track Evaluation

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.

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.

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.

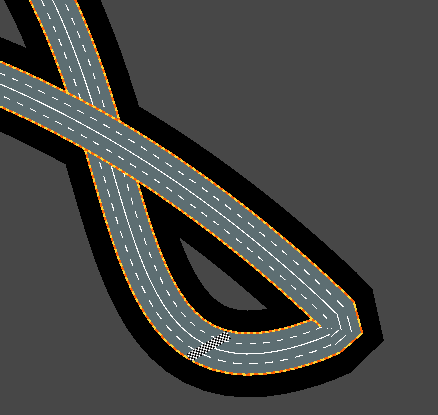


Figure - Example of tight corner

## Player Tracking

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.

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.

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. As shown in Figure 13, tracks that have not been played after 3 days lose half their fitness, this was done to attempt to keep the tracks evolved up to date with the most recent players preferences.

Figure - Time fitness graph

## Ratings and Requests

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.

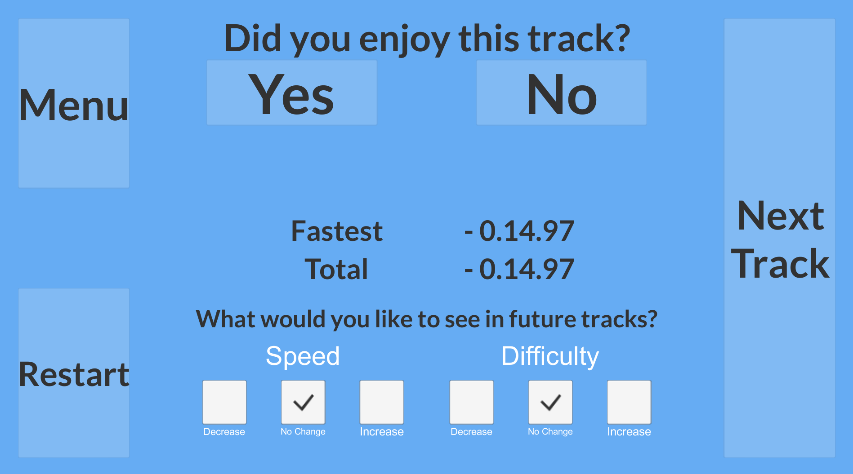


Figure - Post race request prototype

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.

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.

## Evolutionary Algorithm

### Genotype Representation

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.

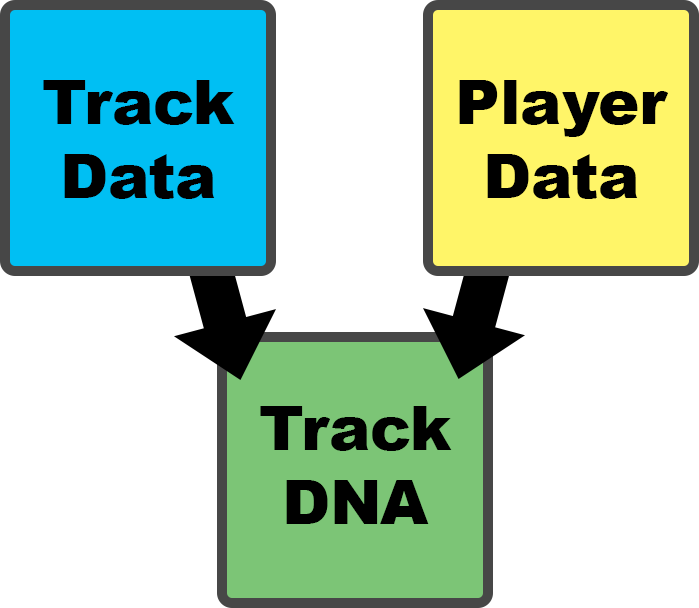


Figure – Track DNA

### Initial Population

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, the resulting TrackDNA is added to the initial population. To create a new track, the initial population needs to be higher than 8, after it has passed that threshold the player is given the option to generate a new track.



Figure - Track progression.

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 (Tversky and Kahneman, 1974).

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.

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.

### Fitness Function

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.

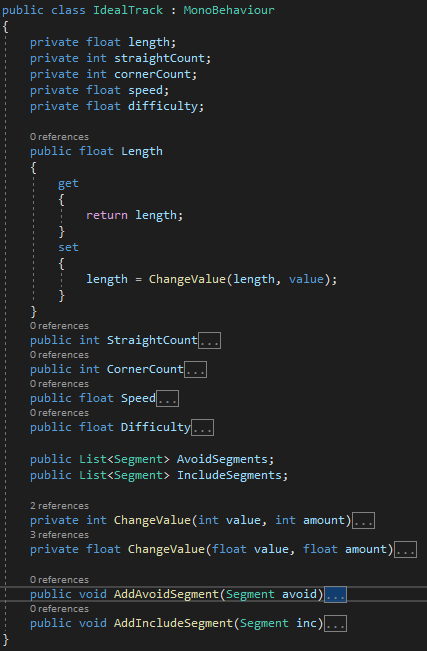


Figure - Ideal Track Code

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.

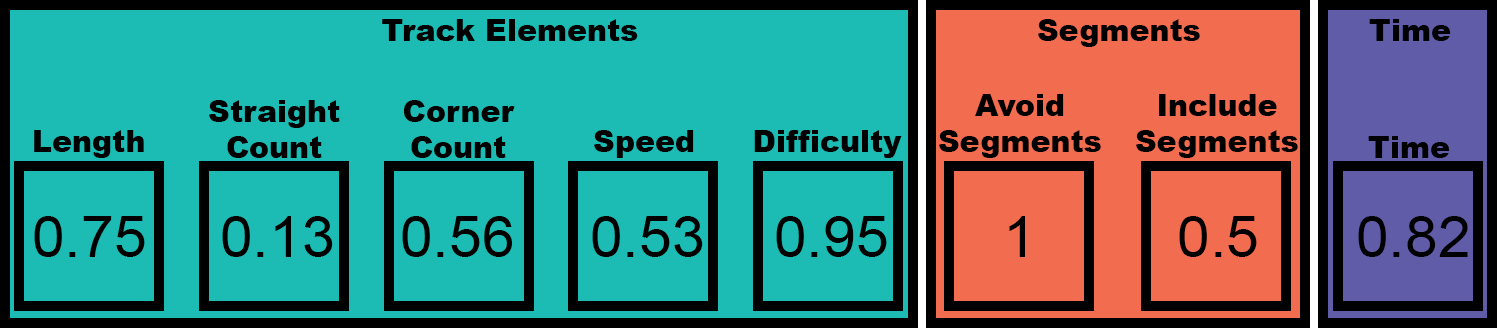


Figure - Fitness vector



Figure - Weightings of the fitness functions. The values were never tested.

### Parent Selection

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.

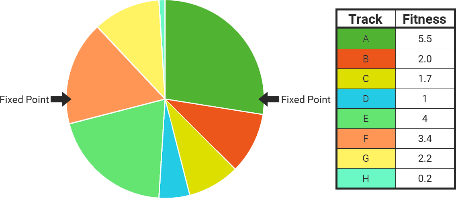


Figure - Stochastic Universal Sampling

### Crossover

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.

The parent scaling would sometimes change the track too much as the equidistant system (also used by checkpoints) 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.

Figure - The same track before and after scaling.

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.

### Survivor Selection

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.

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.

# Self-Assessment of Learning

Evolutionary Algorithm

# References