MSc Thesis

Universiteit Utrecht

Faculty of Science
Department of Information and Computing Sciences

Modeling Race Track Difficulty in Racing Games

Robin van der Ploeg
ICA-3019934

July 8, 2015

Supervisors:
dr. M. Wand
M. van de Hoef, MSc

Contents

Abstract

Recent years have shown a rising popularity of procedurally generated content, such as automated level design. To ensure the player enjoys the content, game developers need to make sure it is suitably difficult. This is challenging if at all possible when the content is generated after the product has shipped. The designers need to make sure the game can automatically alter the variables that control the difficulty, depending on the performance of the player. To determine which variables used in level generation control difficulty, a difficulty model is required. We attempt to find such a model for the racing game genre.

To identify what parts of the track most define the difficulty, we use two approaches. First, a data driven model, which uses machine learning to recognize difficult sections on the track. Second, an analytical model that attempts to predict where cars are most likely to lose traction, following the rules of physics. Using a custom-made racing game, our methods are tested empirically through player testing on various procedurally generated racetracks.

Results show that while we can not perfectly predict all difficult sections of a racetrack, crashes can indeed be predicted with above-average accuracy (over 60%) using simple algorithms, with relatively sparse data. The varying level of player performance is identified as one of the most influential reasons why accurate predictions are very hard to achieve. Further analysis of the data suggests some increased accuracy may potentially be achieved with slightly altered approaches.

Our exploratory work helps game developers identify at least the most problematic sections of tracks. We also believe it can be used as a foundation upon which further work can be based.

1 Introduction

Procedural Content Generation Interest in Procedural Content Generation (PCG) has risen with recent developments in the game industry [23]. Cited as influential changes are improving hardware and rapidly increasing costs for big productions.[1] Content generation algorithms, such as those for generating levels, can keep development costs lower by reducing workload [12].

Many varying methods and implementations of PCG can be found, both in literature and in practice. Some areas of game development have received relatively little attention, such as system and world design, or story. Others, including smaller bits of games (such as vegetation or textures) and terrain have been adopted quite rapidly in recent years [10].

Indie games are most frequently produced by small studios with very few level designers, who use these techniques at runtime to increase replayability. After the introduction of Minecraft in 2009, many "sandbox" and exploration games featuring procedurally generated worlds have been developed, such as Terraria, Cube World and Starbound. Most of these games provide gameplay mechanics that can overcome any issues or unexpected side effects of the algorithm. The terrain might contain mountains that can not be crossed, but the game then allows players to build over or tunnel through the mountain instead. Bigger game studios may also use similar techniques, but are more likely to employ more advanced graphics. This leads to more complicated algorithms in order to avoid game-breaking corner cases, which take more time to develop.

Racing games One specific game genre which has received little attention in terms of procedural generation is racing games. While racing games have been around for decades, only a select few contain any obvious form of PCG. Gran Turismo 5 contains a feature where the game generates a track based on a few settings. While Fuel contains an open world which is generated through procedural techniques, it is not "randomly" generated and will be the same every time. This means it is mostly used as a compression technique, rather than to add replayability. There are also some (but not many) indie games that feature racing in procedurally generated worlds, such as Race The Sun and Infinity Random Race (IRR). Both of these

[1] C Chapple, "Can the game industry keep a lid on rising development costs?", in Develop. May 14th 2014, viewed on 11 June 2015, http://www.develop-online.net/analysis/can-the-game-industry-keep-a-lid-on-rising-development-costs/0192815

games include game mechanics that allow the player to get around any unforeseen issues generated by the algorithms: Race The Sun is a game that is not limited to a track, and IRR allows the player to transform the car into a robot that can fly over obstacles.

We believe that procedurally generated tracks can add a lot of value to racing games, but that it is not trivial to do so correctly. An obvious limitation is that racing games are less forgiving when it comes to terrain smoothness than other types of games. You usually can not easily change the world or go around obstacles while you're playing, unless special features to specifically do so are added (as previously mentioned). But even when taking only racing tracks that are possible to finish into account, difficulty is still an important factor.

To assure our work towards generating tracks and analysing the difficulty of these tracks is useful, we are interested in exactly how difficulty affects the game experience. Research shows that players like to be challenged, but not overwhelmed [15, 17]. One of the most essential terms in this field is "Flow", a theory developed by M. Csíkszentmihályi [19]. It has been used to better understand how players react to certain situations. It can be summarized as follows: when players are not challenged appropriate to their skill level, they either get bored (challenge level too low) or frustrated (challenge level too high). As neither of these will improve the player's experience, we want to be able to adapt the game's difficulty level to be more appropriate for the player.

Measuring difficulty Adapting the difficulty to the player requires two things. First, we need to be able to detect how well the player is doing. This is not the focus of this work, but could in theory be implemented for racing games by looking at how often the player crashes, how much progress they make along the track in a certain amount of time, et cetera. Second, we need to be able to control the difficulty of the environment around the player. In racing games, the most obvious way to do so is to change the tracks to make them easier or more difficult to drive on (at high speeds). But in order to change the difficulty, we first need to determine what variables control the difficulty of a race track. For traditionally designed tracks this can be done by play testing, but as procedurally generated tracks are different every time, this approach is not an option.

This means it needs to be done through computational methods, implemented by the game developers beforehand. To assess the difficulty of a race track computationally, we will need a difficulty model, capable of analyzing the difficulty of a given section of a race track.

To summarize: our main goal is to find a method of predicting difficulty of race tracks, that can be applied in racing games that use procedural generation to generate race tracks. Preferably, we want a method that can easily be applied, without the need of expensive equipment or long term testing, so that it can also be used by smaller game studios.

Structure of this thesis After going through some of the most relevant related work in Chapter 2, we will explore the environment in which we implemented our methods in Chapter 3. We first discuss the requirements we set in Section 3.1, then the potential candidates for test environments we explored in Section 3.2, and finally our eventual Unity implementation in Section 3.3. We continue with our first approach, the data driven model, in Chapter 4, followed by the analytical model in Chapter 5. Both of these sections are subdivided into subsections for the hypothesis, the model, our implementation method and the test results. A comparison between the models is given in Chapter 6, where we first compare the results in Section 6.1 followed by a discussion of the models and their limitations in Section 6.2. Finally, our conclusion can be found in Chapter 7, with future work discussed in Section 7.1.

2 Related Work

As Procedural Content Generation (PCG) in general is very broad, we will only discuss related work on PCG related to procedurally generated racetracks. This is followed by work on suitability and enjoyability. Next we will discuss works on how we can measure such a quality, including difficulty models for games. Finally, we take some time to discuss work (or rather, the lack thereof) that contained methods designed for three dimensions, rather than just two.

Racetracks On the subject of racing games and race tracks in particular, the work of Julian Togelius is cited very frequently. Togelius has written about computationally evolved track design [4], and AI controlled cars using hand-crafted algorithms as well as machine learning methods [3, 11], discussing methods on how to both generate and navigate tracks for racing games. He has also written about more general forms of procedural content generation in games, such as procedurally generating various forms of content including game rules [2], and even generating entire games starting from nothing [14].

Togelius' work has been the base for many other articles such as those by Cardamone, Loiacono and Lanzi [7, 8], in which they describe a framework for interactively generated race tracks using evolutionary algorithms. Wang and Missura [9] also describe procedural generation of tracks, which they approach as a discrete sequence prediction problem.

In [1], Cardamone discusses evolutionary learning (for AI) as well as content generation. He shows techniques that can be applied to racing games, first-person shooters (FPS) and platforming games. Procedural content generation (PCG) in platforming games in particular is something that has received attention from other authors who have written on the subject of racing games, such as Togelius [6]. While our main focus here will be racing games, some PCG techniques used in other game genres can give us additional insights.

Generating race tracks is not our main goal, and is only done to quickly generate tracks for testing purposes, as explained in Section 3.3. Because the methods discussed above did not meet our requirements or were time consuming implement and use, we designed a simple method using A* in combination with splines to achieve this goal.

Suitability and enjoyability Aside from generating content, work has also been done on evaluating it to determine if it is suitable for playing, and enjoyable. One of the people frequently cited on this is Georgios N. Yannakakis. In [15] a simple physical game is adapted based on player performance and preference, which is shown to increase the player's enjoyment of the game. Something similar is then done in [16] for a platformer adapted to incorporate similar preference learning.

To determine if a race track is enjoyable, first we need to define what makes a racing game enjoyable. In [4] (from 2006), Togelius et al. mention that they were *"unable to find any prior research on what makes it fun to play a particular racing game"*. Instead, based on their own experiences and *"opinions gathered from unstructured selection of non-experts"*, they define five points they believe are important:

- The sensation of speed; a high maximum speed.
- Not boring; a sufficient amount of challenge.
- Not too difficult; not too challenging.
- Variety of challenges; not repeating the same challenge constantly.
- Drifting or skidding in turns.

In addition to this, they also offer a different perspective, based on Raph Koster's work [24]. He says that playing and learning are intimately connected. Applying this to racing games, they say that *"a good racing track is one on which the player does pretty poorly the first time he plays, but quickly and reliably improves in subsequent races"*. However, in [5] (from 2007) Togelius et al. also mention these and other hypotheses are just that; hypotheses, with no empirical studies to back them up. We have not found indications of studies on what makes racing games "fun" performed since then.

In both the interpretation of Koster's work and their own list of important factors, finding the right amount of challenge for the player is a core element. To do so, we need to know what makes a race track challenging for any particular player.

We made no further advancements in analysing the enjoyability or suitability of racetracks. Instead, we focussed on methods to determine the difficulty of tracks, in the hope that this in combination with previous work on enjoyability would prove useful to improve race tracks in the future.

Difficulty models To our knowledge, no relevant work on race game specific difficulty models, based on the track, exists. We do know that simple models able to determine how well the player does compared to previous records, other players or AI have been used in racing games. Commonly, in single player games, these are based on how far ahead or behind the player is when compared to AI controlled cars. Notoriously, in older games in the Mario Kart series, the AI controlled characters would speed up quite heavily if the player got too far ahead. This is known as "rubber band AI", and is considered by many players to be bad game design. Naturally,

players are bound to feel cheated if no matter what they do (including skipping big track sections using glitches in the game), the AI will always catch up, then proceed to stay a small distance behind the player. The effect is mentioned by several papers on dynamic difficulty adjustment (DDA) such as by Lopes and Bidarra [13] and Yannakakis and Hallam [15], mostly as an example of what is considered a wrong implementation.

None of the referenced work describes a proper implementation of DDA in racing games, or difficulty models for racing games. Specific difficulty models, such as those described by Moffett [17], generally use variables that are not applicable to racing games (such as "enemies" or "obstacles"). Additionally, we found no racing games that apply dynamic difficulty adjustment to the tracks, only on AI controlled opponents. This leads us to believe that there is not much, or any, related work on (predictive) difficulty models for racing games.

General work on the subject includes a thesis by Jeffrey Moffett [17], which focuses on using causal models for DDA. It describes how to determine player's enjoyment of the game, using a directed acyclic graph based on factors and quantities from within the game and the player. According to this graph, the avatar performance (that is, the part of the game controlled by the player, such as a character or car) is influenced by the environment, the player and avatar abilities, the last of which is in turn influenced by avatar actions and parameters. It is noted that racing games can have a large set of environmental factors which can be manipulated (such as the shape and condition of the track). According to the model, this means the environment has many ways to influence the player performance. Sadly, racing games and environment factors are not described or explored in more detail.

Jennings-Teats et al. [18] detail a model they call Polymorph, which is used for dynamic level generation (for a 2D platforming game, in their case). They note that "nearly all" techniques used by others for DDA are *"focused on basic parameter tweaking, while the difficulty of many games is connected to aspects that are more challenging to adjust dynamically, such as level design"*. As we are looking for a way to dynamically adjust a race track, this implies that most techniques used by others do not apply to our case. This seems to confirm our suspicion that not much research has been done in this field regarding racing games. Additionally, they also note that *"most DDA techniques are based on designer intuition, which may not reflect actual play patterns"*. In an attempt to not have such issues with our models we avoided designing our models around very specific cases, or depending on implementation-specific player abilities, as much as possible.

In Chapter 4 and Chapter 5 we discuss the models we designed, which we believe go beyond the work already done on analysing the difficulty of race tracks.

The third dimension: height One interesting factor of the methods used in most research mentioned so far is that they are mostly limited to two dimensional space. We are interested in methods that can be applied to common types of modern racing

games, many of which are based in three dimensions. We therefore specifically searched for work that also took the third dimension, height, into account.

We found that some works, such as those on Trackgen [7, 8] by Cardamone, Loiacono and Lanzi, or the recent work by Wang and Missura (from 2014) [9] implement their tracks in a 3d environment (TORCS and TrackMania: Nations Forever, respectively). However, it seems that this is only the final implementation, as the TORCS tracks appear to be flat and the TrackMania tracks only change height where they need to, for example in the case of self-crossing tracks. Height does not seem to be taken into account before actually generating the physical track: illustrations and algorithms all seem to imply the methods used work in, and were developed for, two dimensional space. While this does not mean they can not be used in three dimensional space, it might mean that height differences are not taken into account in any of the models described so far. In fact, in the "Simulated Car Racing Competition" API for (TORCS) controllers as mentioned by Cardamone in (Table 5.1 in) [1], height of any point of the track, the car, or any opponent cars does not even seem to be available in any form to those who would want to use it.

Being limited to 2d space means that applying these methods to certain games within the racing genre will be challenging. Many games, in particular traditional racing games on race tracks, such as those about Formula 1 racing, have most action taking place on a very flat road. Other games, such as those focussing on illegal street racing or off-road rally driving, combine sharp turns with height differences. The lack of control while cars are airborne often plays a big part in the experience these games offer, and we believe it should not be overlooked. However, we could not find any related work that includes this factor. Nevertheless, we aim to develop a model that can handle all three dimensions, rather than be limited to just two.

3 Test Environment

Based on our initial research (see Chapter 1), as well as our search for related work (detailed in Chapter 2), we concluded that we did not know (nor could we find) much about computationally analyzing what made racing tracks fun, or difficult. As we believed this to be a crucial part of procedurally generated race tracks, we wanted to find a difficulty model. To test the validity of any difficulty models, a testing environment was required. The requirements we set for such an environment are described in Section 3.1. We tried to find existing racing games meeting these requirements, which is described in Section 3.2. We finally decided on implementing a simple racing game using the Unity game engine, as described in Section 3.3.

3.1 Requirements

With very little similar work to base our requirements on, we looked at what we wanted to achieve, and tried to set our requirements based on that. Our goal is finding a method that can be applied in real racing game development. For this we need a racing simulation accurate enough to be like typical games in the racing genre, to help ensure our methods are applicable in practice. This means that ideally, we use an already existing racing game.

We need many different tracks in order to test various cases and ensure our methods are not limited to one specific corner case. As designing many different tracks is very time consuming, doing this by hand would limit the amount of tracks we could produce in a limited amount of time. As such, it is preferred to be able to easily generate these tracks computationally, rather than having to create them by hand. As mentioned in Chapter 2, most related work we managed to find limits itself to two dimensions. We decided that to check if height differences made much of a difference in the models, some level of control over the way tracks are generated is needed.

Finally, the models we designed and our chosen implementations added some extra requirements for the environment which are not commonly found in racing games. As explained in the sections on our methods (Section 4.3 for the data driven model and Section 5.3 for the analytical model), we would have to gather data about the player's performance. This means it is important to have some logging functionality, to find if and where exactly players leave the track ("go offroad" or "crash"). In addition, our models require information about the shape of the tracks (most importantly,

curvature and banking angle). This data also needs to be available and accessible, either from logs generated by the game or directly from the track specifications.

3.2 Considered environments

3.2.1 TORCS

As mentioned in Section 3.1, using an existing racing simulator would be preferable for various reasons. In particular, as we aim to design models for use in fully functional games, using such a game as our test environment would help ensure we meet this requirement. Due to the logging functionality requirement, which is clearly not generally considered a part of standard racing games, a commercial products with the required features could not be found.

Previous research work involving racing simulations, such as that by Togelius and Cardamone, was performed either on simple hand-made simulations [4, 5], or open source software [7, 8]. The most important of the open source simulators used seems to be TORCS: The Open Racing Car Simulator [1]. Its modularity and extensibility are cited as important factors for its use in research. TORCS has AI racing as a focus, and while human players can control a car themselves, computer controlled cars (also known as "robot" or AI cars) is one of the strong features of this simulator.

Limitations One of the features of TORCS that stood out right from the start was the big focus on AI racing. While not an issue in normal situations, it turns out that the AI is easily tricked by placing “ramps” in the track. A player could simply drive over the ramp, across a big gap, and land on the other side with relative ease. The AI, however, would act like the height difference did not exist and usually crash. This was also reflected by the AI API used for TORCS bot writing competitions (as explained by Cardamone in [1]), which we believe lacks any feature to check for height of points on the tracks. As we were aiming at finding models that could be used in a wide variety of racing games, no support for height differences was a big limitation.

While we might have been able to modify TORCS (as it is open source) to include this support, we would also have had to write a custom AI that could handle the height differences. And, as all the existing tracks were mostly flat, we would have needed to either design tracks by hand, or write code to convert procedurally generated tracks to TORCS’s track file format. Due to a lack of knowledge of the existing codebase, this was all a big risk. Even if it was possible to modify the code, it would at least take a long time to read and understand the existing code well enough to modify it correctly.

[1] http://en.wikipedia.org/wiki/TORCS

3.2.2 Speed Dreams, VDrift and other free / open source projects

Speed Dreams is a similar open source project, which started from the same source code base as TORCS but deviated from it in some ways. In particular, it shifted the focus from AI to user-oriented racing. Similarly, a different open source racing game project called VDrift was also available, based on an older project called Vamos.

Speed Dreams and VDrift were both considered as options, but did not contain any additional features useful for this research project when compared to TORCS. Their lesser popularity in the scientific world, combined with a seemingly smaller community, made us discard them as candidates for testing.

The few other similar games that were found were not considered for this research, mostly due to lacking features or support.

3.2.3 Unity / Custom-made racing game

Unity is currently one of the most popular game development engines amongst smaller development teams [2]. Recently it has even been used by some big developers such as Blizzard (Hearthstone: Heroes of Warcraft) and Ubisoft (Grow Home) [3]. Previous experience with it had proven it to be simple and effective, with a very large and active community. This community offered a multitude of tutorials, freely available code and assets, and many answers to racing game related questions. Unity itself offered a physics engine, even including built-in support for more complex things such as wheel colliders, to simulate the physics related to car wheels such as suspension and friction. Earlier research on procedural terrain and track generation (see Section 3.3) had already been implemented in Unity. This made it a very simple task to take some example car implementations and instantly be able to drive around on our already generated tracks.

These basic car models were not as advanced or sophisticated as that of TORCS. But with only some small changes it soon felt like a real game, and making further changes proved very easy and quick. Implementing other required features, such as logging capabilities, were also no issue. Using Unity also allowed us to easily implement parts of our analytical model (see Section 5.3) directly in the game. This meant exporting internal values was not needed, so we could keep the output for further processing and analysis relatively small.

[2] https://unity3d.com/public-relations
[3] http://en.wikipedia.org/wiki/List_of_Unity_games

3.3 Unity implementation

For our case the simplicity and flexibility of Unity, combined with previous experience and previous progress in it, outmatched the lesser detail of the car model. In fact, the Unity car model being more basic (and thus less specific) could be seen as an advantage if we want to keep our models valid for a wide range of race games. It was accurate enough to feel like a racing game, not show any erratic behavior, and (despite a lack of proper gameplay features) was even declared to be "quite enjoyable" by testers. We decided to expand upon what we already had and implement our test environment in Unity, as detailed below.

3.3.1 Terrain generation

To quickly generate many different landscapes, we wrote some code that edited a heightmap, and applied it to a terrain object in Unity for quick visualization. Texture blending with 4 textures was applied based on height and steepness of the terrain, to improve the visuals and further clarify potentially problematic areas. As it was not our main focus, we settled on 3 layers of Perlin noise, with varying scaling and power values, and left it at that. This generated big, smooth mountains, smaller yet slightly more irregular hills, and yet smaller but steeper bumps. While we could have added further detail, we found it unnecessary as placing racetracks on the terrain would smooth the terrain, removing these details. To make the valleys bigger, we raised all the values on the heightmap below a set minimum to that value. This was done to increase the amount of straight roads, to get more variation between curving and straight roads. Finally, in preparation for our tracks, we decided that the terrain was too rough. This can be seen in Figure 3.1. To fix this, we increased the resolution of the heightmap to make the hills smoother.

3.3.2 Track generation

As our next step, we wanted to see if we could easily generate roads without ignoring the terrain. The road should follow the terrain smoothly. We did not want to go from one point on the terrain to another in a straight line, building bridges over valleys and tunnels through mountains, as it would create boring tracks.

To do so, we implemented a basic path finding algorithm, calculating paths between points on the heightmap. Each point on the heightmap represents a node, connected to the 8 surrounding nodes in a grid. We tried using A* as our path finding algorithm, but found it too slow on large maps. Eventually we settled on an implementation of Dijkstra's algorithm, using a priority queue for the nodes, which performed better. The cost function was mostly the distance in the horizontal plane, but heavily influenced by height differences between points on the heightmap: the

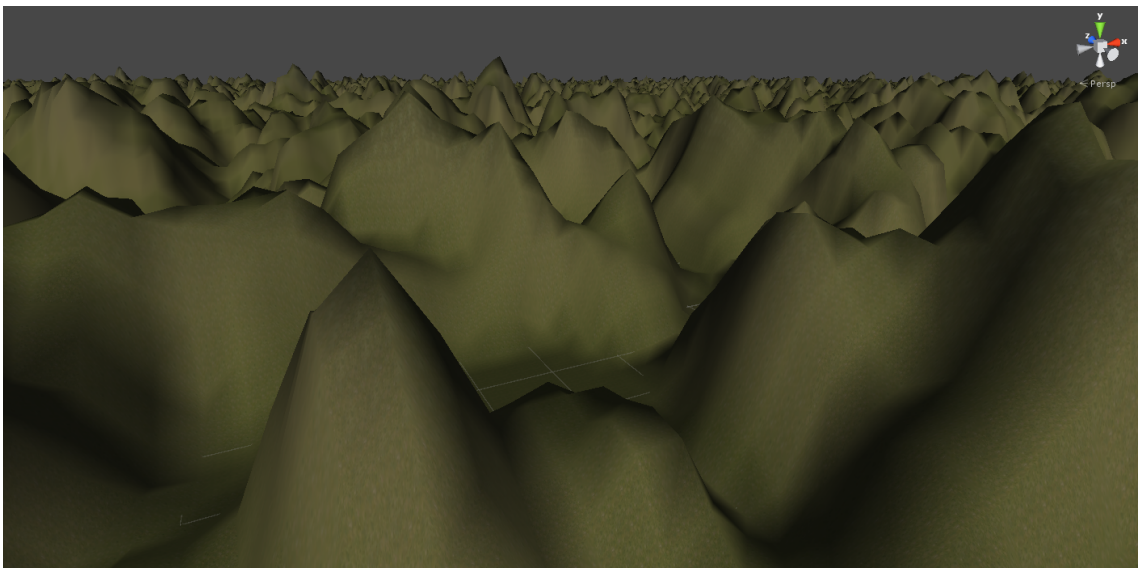

Figure 3.1: The terrain as generated by our method. The terrain depicted is using a lower resolution heightmap than our final version.

difference in height is squared, then multiplied by a variable depending on the resolution of the height map. This caused found paths to often snake around mountains, rather than straight up and down them, in an attempt to have as little height difference as possible unless absolutely necessary. Roads showing similar behavior can be found in real-life mountainous areas around the world.

Because the height difference penalty was so high, the road would sometimes fold over on itself in zigzag patterns while going up steep mountains. This effect can be seen in Figure 3.2. Since this would create undrivable roads, we wanted to prevent this from happening. These problematic areas always contained many sharp corners, so we attempted to fix the problem by reducing those. We gave extra weight to the next node to explore based on the angle with the node the current node was reached from. For instance, if a node was reached from the node to its north, the node to its south would get no penalty. But the nodes to its south-east and south-west would get a small penalty, and the nodes to its west and east would get a bigger penalty. We made it impossible for the algorithm to take very sharp corners, that is, go to nodes next to the node it was reached from in the algorithm. In the above example case, it could not choose the node to its north-east or north-west as next node to explore from that node.

Refining the path The paths found by the path finding algorithm, as they are limited by points on a grid, were very coarse. Taking the points on these paths as control points, Bézier splines were calculated to find the basic shape of the road. This removed the sharpest corners from the line of points found by the path finding algorithm. Based on how detailed the road needed to be, a varying amount of points

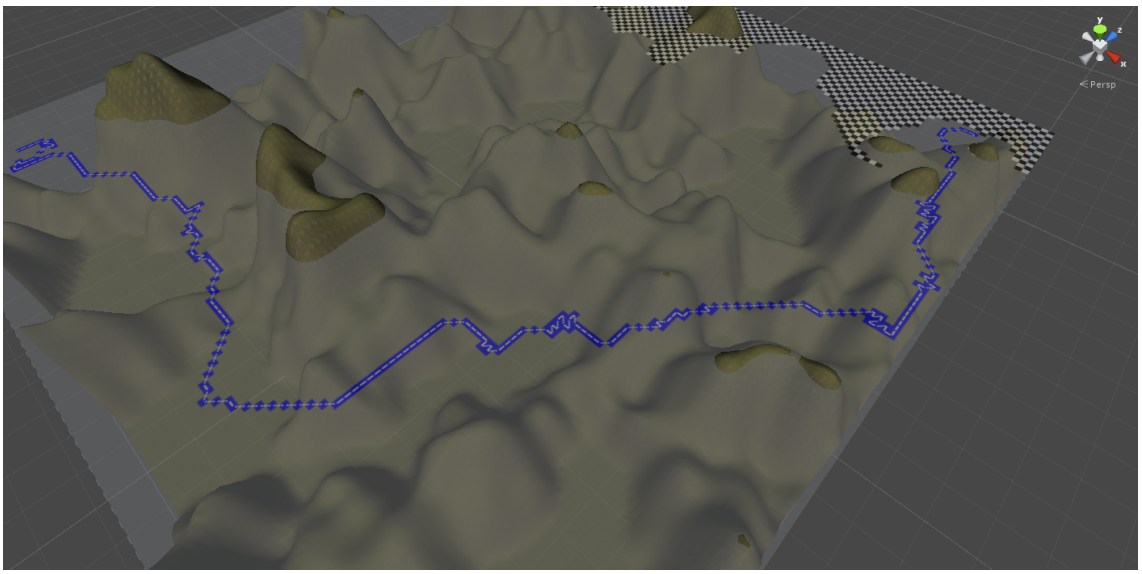

Figure 3.2: A* path finding implementation without penalties for sharp corners, projected on a transparent plane over the terrain. Blue squares indicate the found path, with white lines to indicate the direction to the next and previous square in the path. Note the zigzagging pattern of the path where the road moves up and down hills.

along these splines were used as control points for a cubic spline. This depends on the density of the heightmap, and the scale of the road and cars, so it varies per implementation. The terrain below this spline was smoothed out slightly, to remove uneven terrain coming through the road. The road geometry was then formed by extruding the spline in two directions to generate the vertices, with the banking angle based on the heightmap. On sides of hills that were fairly steep even after smoothing the terrain, this would sometimes generate roads that have a "negative" banking angle. In other words; with a mountainside to the right, a road with a curve to the right might actually have the left edge of the road be lower than the right edge. Clearly, this is not what we can expect in real life, as it makes cars more likely to lose grip on the road, and crash (see Section 5.2 for more information on the related physics). However, it does not make the road undrivable, only more difficult to drive on. Therefore, we did ensure this would not happen, and treated it as an interesting case for our models to work with.

3.3.3 Racing game

As described in Section 3.2, after comparing different race game implementations we settled on creating one ourselves in Unity. In the end, our control model included basic steering, throttle, braking and hand brake. Unity has a physics model built in, as well as a wheel collider (which functions as a collider for the wheels with the road,

and includes many variables for suspension and friction). The terrain and tracks (and fitting colliders) were used as described above. We had no moving objects in the game other than the car, no other cars, and no obstacles or decorations.

Most values for physics and the car's controls, such as the values for friction, suspension, weight and size of the car, were set as close to real-life equivalents as possible. Some of the more technical values amongst them were kept at the recommended defaults in the Unity physics engine, unless different values were preferred for racing games. A few were changed slightly to more accurately act like an arcade-style racing game rather than an accurate simulation. In particular, changes were made to handling of the car and the maximum turning angle. As this higher maximum turning angle made the car far more likely to flip over in corners, the center of mass was placed unrealistically low to compensate. This made it harder but still not impossible to flip over the car. In addition to these, gravity was doubled, as the default value produced unrealistic results despite making sure all other values were set correctly. It seems this is an issue with Unity, as other users have reported similar issues[4]. However, it is also possible that some variables changes were counteracting each other, especially in an engine as versatile as Unity.

Figure 3.3: A screenshot of the final game implementation.

[4]http://answers.unity3d.com/questions/395618/gravity-seems-to-be-too-slow.html

4 Data driven model

With our test environment set up (see Chapter 3), we were ready to start modeling and implementing difficulty models. Based on our related work, we knew that machine learning could be used to construct difficulty models, which is why we started working on a data driven model. Our hypothesis is described in Section 4.1, followed by an explanation of the model we designed in Section 4.2. We describe our implementation method in Section 4.3, and finally report the results we found in Section 4.4.

4.1 Hypothesis

Mentioned in our related work is the use of machine learning for defining difficulty models. As shown by Jennings-Teats et al. [18] it is possible to label control parameters of level generation for a platformer game. Using machine learning you can then assign them weights based on their influence on difficulty. They note that while their implementation is for a platformer, the same principles can be applied to other genres. As they say, the following tools need to be available: *"data collection for how difficulty is impacted by combinations of level components —including structural differences— and a level generator to create short segments that can be strung-together in real time into playable levels"*. We believe the latter can be achieved for racing games. Some popular games such as the TrackMania series allow players to string road pieces together in a simple editor. So, possibly given some limitations, we believe stringing together pieces of tracks can be done by an algorithm in real time.

This leaves us the task of collecting, as Jennings-Teats et al. put it, the data on *"how difficulty is impacted by combinations of level components"*. Jennings-Teats et al. used machine learning for this, and we made an attempt at doing the same.

4.2 Model

First, we decided on what we wanted to achieve with our machine learning method. The goal here was to find not the exact reason why a section was difficult, but to simply determine if it is difficult or not. Given some input, our method should be able to make a usually correct prediction on whether or not the player had a high

risk of ending up next to the track. We chose to use a support vector machine (SVM) implementation, as it is a popular and often used binary classifier.

To keep our models simple, we limited our roads to a non-diverging road with no obstacles, and no cars except for the player. We also limited our roads so that when projected on the X-Z plane (with the Y axis being height, so top-down), it does not self intersect. Simply put, this means the road never goes over or under itself.

As input for the SVM, we chose to use the shape of a section of road. The idea behind this is that sharp corners are more difficult to drive than straight roads, and sloped roads again more difficult than flat ones. To train the SVM, we also needed to include if such a section was indeed dangerous. Because one individual point on the road does not give any relevant information, we would need information on a continuous section. We decided to limit our input to just the positions of points along the middle of a section of road, in 3D space. Other options we considered were the first and second derivative of these. This would be the vectors between sequential points, and difference between sequential vectors between sequential points, respectively. As the derivatives can be calculated from just the positions, we decided to use those.

In order to detect patterns, without being too diluted by variables that would did not have a big influence, we needed to either keep the amount of variables limited or gather very large amounts of data. Due to limited time and resources for our testing, we therefore decided not to include the banking angle. Another reason to use only the positions is that it is a very general case, rather than applicable to just one specific game. As mentioned by Jennings-Teats et al., most methods for DDA are designed on designer intuition, rather than actual play patterns. This is something we wanted to avoid, as we are specifically trying to find a model that can be applied to any racing game, or at least a large section.

4.3 Method

Our simple racing game implementation constructed a road procedurally based on points on a spline. These points, more or less spaced out evenly along the spline, were used to generate the road geometry. However, as they were an ordered list of points in the middle of our road, they quite accurately represented the entire road. This meant we could also use them as input for our data driven model. This list was cut into overlapping pieces to create representations of road sections. For each point in the list, that point combined with k points in either direction would form one section. In our implementation, we went with an initial value of 4 for k, to get sections of 9 points long.

4.3.1 Flagging records as off-road

Our race game did not have a border next to the road, so rather than crashing, players would end up next to the road if they missed a turn. In order to train the SVM to find difficult sections, we needed to know where this would occur. For this purpose, the raw data on how the road was shaped needed to be combined with actual player data. To achieve this, we simply drove along each generated road, and marked the positions where the player left the road.

We then searched for the closest point within the set of points representing the road, and marked that point as "off-road". Each section in which that point was present, rather than only the one for which it is the center point, would be marked as off-road entirely. This was done because players might lose control at a difficult point, but only leave the actual road slightly further on. It could also be that a player would see a difficult section coming and try to prepare, but overshoot and end up next to the road before the hardest part of the section was reached. As such, limiting the event to only flag one section as off-road might cause the actual cause of the event to be overlooked. Easy sections which just happened to follow a difficult corner causing an off-road event could thus also be often flagged as off-road. We did not expect this to pose a big problem. As there are many similar easy sections which generally do not contain off-road events, the SVM should qualify the few that are marked as such as outliers.

One important detail here is that we only drove over each road once. We chose not to include separate races on the same track as separate data to avoid overfitting the SVM, due to limited amounts of data.

Defining data records With the data combined, we now had a list of records for each road we had generated and driven on. One record would be $2*k+1$ positions in 3D space, or $3*(2*k+1)$ floating point numbers, plus one Boolean value indicating if the player had ended up off-road somewhere in that section. For instance, a value of 4 for k would make one record 28 values long. The length of the list of records depends on the length of the road and the value for k: for n points in a road, the length of the list of records is $n - 2 * k$.

4.3.2 Invariant position / rotation

Positionally invariant For our purposes, an identically shaped road or corner in two different locations can be considered identical. That is to say, their position in the world does not matter for our difficulty model. Two identical road sections, one starting at the origin in our game world (0,0,0), and one at (2000,2000,2000), should be treated by the SVM as equal. To make sure that the overall position of the points would not influence the SVM, we simply translated each road section by the inverse of the average position of all points in that section. The middle of every

section is thus placed at the origin. This was an easy solution to make the data positionally invariant. A two dimensional version of this is shown in Figure 4.1.

Rotationally invariant However, we also wanted to make our sections rotationally invariant. Two completely straight sections of road, one pointing North to South, another East to West, should also be treated as identical by the SVM. To achieve this, we took the angle θ of the average direction of the entire section, and rotated all points around the midpoint by the inverse of θ. As we had already translated each section, the midpoint would always be the origin. To calculate θ, we first needed the average direction. For a set of vectors, this is normally calculated by adding up all vectors in the set. However, as our points were already in order, this adding up the vectors from point to point would equal following the road. Thus, we could simply take the vector from the first point to the last point in the section. The average angle was then defined as the angle between the normalized average direction and a target vector (1,0,0). Simply rotating every point around (0,0,0) by the inverse of this angle produced the desired result. This process is shown in Figure 4.2.

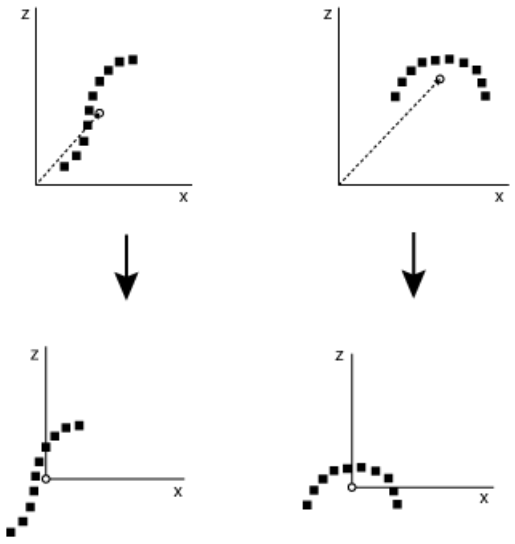

Figure 4.1: Making the road sections positionally invariant

4.3.3 Cropping the data to exclude the ends

While we now had an invariant list of records, there was still one easy way to reduce the amount of outliers. Our roads were generated based on a spline through points, which in turn were generated by a path finding algorithm. Due to the simple

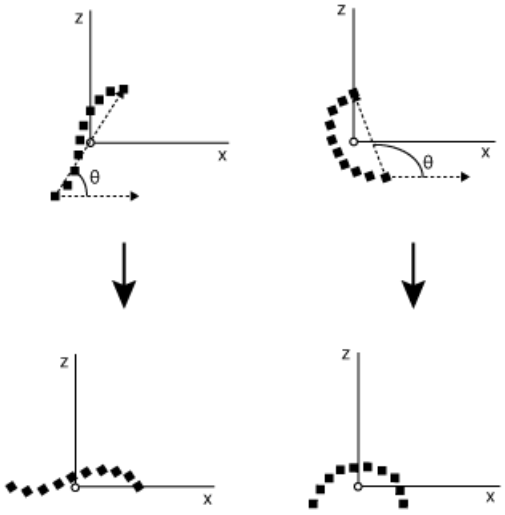

Figure 4.2: Making the road sections rotationally invariant

implementation, the very start and end of the road could be very sharp corners, and hard to drive on. These pieces of road would probably create off-road events if driven on as usual. However, as in our test environment the player starts a little bit down the road, the very first section of road was never driven on. This meant that this section, unless the player drove backwards, would never produce an off-road event. So the road sections at the start of roads which should often produce an off-road event almost never would, consistently creating outliers. Similarly, the car was not automatically stopped at the end of the road. Usually, the very end of the road was not driven on if the player saw it ahead of time and stopped. Otherwise, the car would drive off the end of the road and generate an off-road event, even on an (easy to drive on) straight road. Either way, this would create outliers.

Reducing outliers To make sure these outliers would not spoil our data, we simply culled the first and last few records on each road. As the amount of outliers created by these events at the very ends was based on the section length, and the length of sections was based on k, we based the amount we culled on k. We went with $k+1$ in our implementation, which is exactly the amount of records marked off-road when driving off the very beginning or end of the road. They represent the closest point to the off-road event, plus k points in the direction of the rest of the road, away from the end. Based on early test results this seemed to cover all outliers and not much more. Additionally, we noticed that there were never off-road events at the very start of the road. We concluded that this was due to the low velocity of the car right after starting. As the velocity of the car is not known to the model, these data

samples skewed the statistics. To compensate, we culled a few more records at the start. We made an estimation of how many points would be ideal based on early test results. We settled on 4, but this of course varies per implementation, based on the distance between points as well as acceleration of the car.

4.3.4 Feeding the data into SVM

To find out if we could predict off-road events, we used our gathered data as input for an SVM implementation. We used R [20], with the RStudio IDE, and the e1071 package [21] for its SVM functionality. Additionally, we wanted to draw Receiver Operating Characteristic (ROC) curves, which are visual representations in a graph of false positive rates on the horizontal axis versus true positive rate on the vertical axis. They can be used to get a first impression of how well a model performs in identifying positives. To generate ROC curves, we used the ROCR package [22]. It featured parameters that could be tweaked for best results, including several different kernels, and the option to allow for prediction probabilities. As we mentioned in Section 4.2, the goal here was to find out if a road was dangerous, or rather, if we could correctly assess a road's difficulty using this method. For our purposes we did not need to know why a section was difficult, only that it was or was not. As such, we wanted our SVM to just output a Boolean value for each section ($3 * (2k + 1)$ floating points), representing a prediction for an off-road event. We could then test this against the results of a player actually driving along the road the predictions were made for. A prediction for an off-road event on sections where the player actually went off-road is a true positive, those where the player did not drive off a false positive. Likewise, sections where no off-road event was predicted were marked as negatives, with those containing an actual off-road event marked as false negatives and the rest as true negatives.

4.4 Test Results

4.4.1 Results

Our results are detailed in Table 4.1. They are visualized in Figure 4.3, which shows the ROC curves of our results. The curve is formed by varying the probability threshold used to predict off-road events between 0 and 1. A probability threshold of 1 accepts only results that the SVM believes are 100% certain to contain an off-road event, which is usually none, resulting in no false positives and no true positives. This is the lower left corner of the graph. Likewise, a threshold of 0 accepts everything, resulting in all possible true and false positives; the top right corner of the graph. The AUC column in our table stands for "Area Under Curve", which can be seen as a measure for how well our model performs.

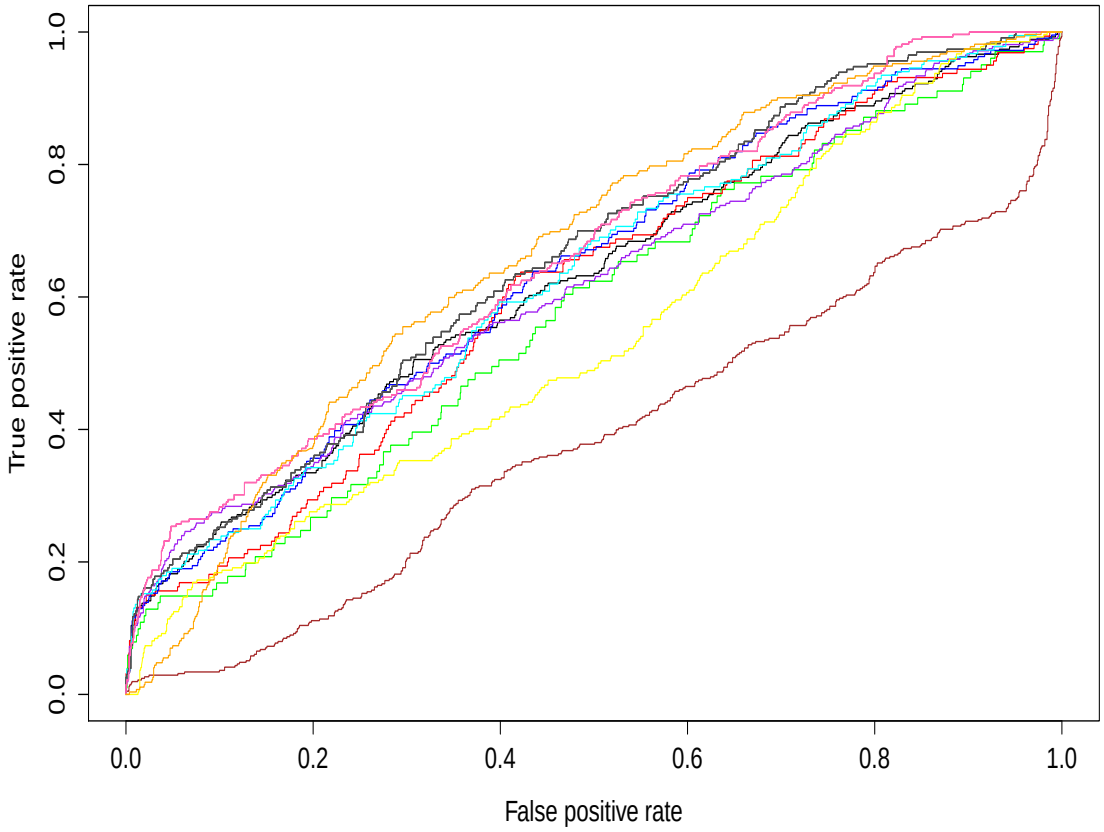

Figure 4.3: ROC curves for the results of the data-driven model.

Splitting the data Due to our limited amount of data, we were limited in our options on how to split the data into a training and testing set. We considered using 80% / 20% for training / testing respectively, but with our amount of data this would make the amount of test data too limited. Resampling methods (such as Jackknifing) were too time-consuming to apply with the many different variables we wanted to compare. We thus decided to split the data evenly in half to give us our training and testing data. We started out with a value of 4 for k during testing, which turned out to perform quite well. As our results show, it performs slightly better than $k = 5$ using the full training set. Using $k = 3$ performed even better with the full data set, but using even lower values for k decreased the AUC values. We tried values for k above 5, but they were significantly worse.

Influence of data set size We also tried to reduce the amount of testing data to see if this would influence our results. As it turns out, using less data for training and testing improved our results by roughly 0.03 in the case of $k = 4$, and more

Color	train data / test data	k	AUC	SVM parameters
Green	50% / 50%	1	0.5774019	$\gamma = 1/9$
Red	50% / 50%	2	0.6110372	$\gamma = 1/15$
Blue	50% / 50%	3	0.6365064	$\gamma = 1/21$
Black	50% / 50%	4	0.6261912	$\gamma = 1/27$
Purple	50% / 50%	5	0.6179377	$\gamma = 1/33$
Brown	50% / 50%	7	0.3847989	$\gamma = 1/45$
Cyan	40% / 40%	3	0.6285448	$\gamma = 1/21$
Gray	40% / 40%	4	0.6558581	$\gamma = 1/27$
Pink	40% / 40%	5	0.6587542	$\gamma = 1/33$
Yellow	40% / 40%	5	0.5411106	$\gamma = 1/33$, kernel = linear
Orange	40% / 40%	5	0.6649638	$\gamma = 0.1$, cost = 10

Table 4.1: Details for the ROC curves in Figure 4.3. SVM parameters are cost = 1, kernel = radial, unless otherwise specified.

than 0.04 for $k = 5$. As more data should generally increase accuracy rather than decrease it, unless the data contains incorrect information, this suggests the removed data contains outliers. Varying k for this reduced set of test data shows that $k = 4$ performs quite a lot better than $k = 3$, but very slightly below $k = 5$. To keep the graph and table clear, the results on the reduced data set for k values above 5 and below 3 are not listed, but they show patterns very similar to that of the full data set.

Tweaking SVM parameters After varying k and the amount of data used for training and testing, we tried changing the parameters of our SVM. The default SVM kernel in the used implementation is the radial kernel, but it supports several other kernels. Changing the kernel resulted in lesser results. For instance, a linear kernel with $k = 5$, 40%/40% training/testing data and default SVM parameters resulted in a AUC value of only 0.5411106. This is significantly lower than the result we got using the radial kernel, which used otherwise identical parameters and gave us an AUC value of 0.6587542.

Our next step was to tweak the other parameters used by the radial kernel. We did this by applying a tuning function to the model, and using what was determined to be the best results for the γ and cost parameters. The resulting ROC curve was a bit different from the others in that it started with more false positive than true positive results. After that, it performed slightly better but not significantly so, resulting in an AUC value roughly 0.006 higher than our previous best result. We concluded that tweaking the parameters did not make much of a difference for this model.

4.4.2 Analysis

From our results, we conclude that very low k values, as well as those too high, decrease the results. We believe that values that are too low, resulting in road sections that are very short, simply do not give enough information about the shape of the bend. An off-road event might have happened slightly further down the road from the actual hardest part of a corner, which may not have been included in the section with a low k value. The opposite, a high value for k, resulted in long road sections. This decreased the clarity of where exactly on the section an off-road event took place (or might have taken place, in the case of predictions). To still achieve good results, a very large quantity of data would be required to more accurately train the SVM. We did not have the time and resources for this, and while others might, the need for such large quantities could be seen as a bad requirement for a model to have. Our goal was to find a model that can be applied in practice by game designers, in particular for racing games using PCG to generate tracks. As one of the reasons to use PCG is to save time on designing the levels, a time and resource intensive method is clearly counterproductive considering the intended purpose.

Choosing parameters Finding the right value for k went hand in hand with finding the correct spreading of points along the road. Placing these points too densely resulted in a lot of redundant data, as the shape of the road could be understood from a subsection of the points. Feeding longer or more densely sampled road sections into the SVM resulted in more variables, and thus longer calculation times. As can be seen in the results, increasing the amount of variables did not result in much better results. Placing them too sparse could result in a loss of important data, such as sharp corners, in between the points. The distance between points was based on the amount of detail in the road. Roads with many small bumps and very sharp corners would require a much smaller distance between points than roads that emulate highways, for instance.

4.4.3 Conclusion

Player performance factor One conclusion from the AUC scores was that this model would not be able to pick up every single problem, and would often be wrong. However, it was to be expected that the SVM would make incorrect positive predictions, as the skill level of players varies wildly. A player's performance is a significant factor in what corners they consider difficult, as well as how the SVM is trained. Even players of similar skill levels often have different techniques, or even specific corners, they struggle with. Players that are even slightly proficient at racing games will not crash at every single corner with an above average difficulty level. As such, even if the SVM correctly declared a corner as difficult, a player that drives well (or slow) enough would probably stay on the road. Because we only included one race per track in the data (see Section 4.3.1), this meant that the corner would thus

be flagged as a false positive, even if most players might have went off-road there. The reverse situation, where players go off-road on easy sections of road, also works against the model. Because these outliers are not detected and removed before the data is inserted in the SVM, it will generate many false negatives, reducing the true positive rate.

Fine tuning the SVM The inaccuracy of the model also explained why changing other variables and parameters did not seem to influence results significantly. We believe this was because the model itself was far from accurate. Fine tuning the implementation of an inaccurate model did not result in much added value, as it was being tuned on the inaccuracies as well as the actual valuable data. As long as the values were not too unusual, changing the parameters would result in slightly different decisions in the SVMs feature space, in particular in the area with very mixed truth values. Since this area contained so many outliers in both directions, changing the separating hyper plane would incorrectly classify roughly the same amount in one direction as it would correctly classify in the other, roughly balancing the final result.

Conclusion Our aim was finding a model that could be used to detect difficult (sections of) race tracks. As the model is now, in particular due to the black box aspect of SVM, we found it very hard to use for its intended purpose. The large amount of variables makes it hard to analyze what the SVM conclusions are based on, and we have no easy way of checking if the results are correct. On top of that, because the SVM only returns classifiers, we do not know how to change a road section to either increase or decrease the difficulty.

This does not mean that we believe that machine learning, or SVM in particular, are unfit for this purpose. Our input data is known to contain outliers, and the model is based on a very basic model of the road (only the basic shape, not even including the banking angle). Despite this, the SVM managed to produce output that we consider to be correct more often than not. This is particularly interesting because crashes are very unlikely events to begin with. Because of this, we believe improving upon the known shortcomings of the model might result in a method that could be better used for its intended purpose. In particular, we believe that reducing the amount of input variables could help, as it would reduce the required amount of training data as well as make analysis of the results easier.

5 Analytical model

With our data driven model, described in Chapter 4, we used machine learning in an attempt to train a computer to predict which tracks, or track sections, are difficult for the player. We limited ourselves to just the shape of the road, without making further assumptions on what kind of sections were difficult, or why. This keeps the model generic enough to easily apply to many different types of racing games. But it also limits us: it means we can not easily steer our data driven model towards situations of which we are quite certain that they are difficult. So, for our second model, we decided to use this knowledge by analyzing the road and its expected effects on the player's performance. We call this our analytical model.

As with our data driven model, our analytical model will be explained starting with the hypothesis (Section 5.1), followed by the model (Section 5.2) and our implementation (Section 5.3). After this, our test results are described in Table 5.1. Finally, we dedicated a section on a possible alternative or additional approach in Section 5.5.

5.1 Hypothesis

Ideally, our difficulty model is as simple as possible, while still producing accurate results. Preferably, we want to have some small set of road data that we insert into a function. That would then tell us what sections of the road are difficult to race on as output. The function would need to be adaptable, so it can be reused in other games, on other tracks, or with different cars. While games are not fully equivalent to real world, many years have been spent to make certain aspects as realistic as possible. In particular, modern racing games often have physics models that simulate real world physics quite well. This often counts even for non-realistic racing games, although with different values for things such as gravity or friction. We expect that while driving a real car is different from driving a car in a game, a challenging situation in the real world will also be challenging in a game.

Dangerous situations in real life We want our analytical model to be built around factors that can easily be quantified. To do so, we focus on the road, in particular the interaction between the car and the road. To keep our model clear and concise, we have removed other frequent causes of accidents. For instance, we have no obstacles or vehicles on the road. In addition, the surroundings in our testing environment is limited to mostly smooth hills, which means most turns in the road can be seen

from relatively far away. This lack of unexpected challenging situations helps limit the influence of the player's reaction speed on the performance to a minimum.

With outside influences limited to a minimum, we look at how the road itself influences difficulty. Almost any race track can be easily driven on at a low velocity, and become more difficult to drive on as the velocity of the car increases. If drivers try taking corners at a higher velocity than what is considered the maximum safe velocity of that corner, the risk of losing grip and crashing is high. However, some tracks track sections become difficult at a lower velocity than others. Corners that are considered difficult will generally be driven through at lower velocities than corners that are considered easy. This means that the maximum safe velocity can be seen as a measure of difficulty. Or more specifically, the maximum safe velocity before the car loses its grip on the road.

Without doubt, very sharp corners are harder to drive on at a high velocity than a long straight stretch of road. Cars will lose grip and end up next to the road if they try to take corners at very high velocities, with sharper corners allowing for lower maximum safe velocities. There are some exceptions to sharper corners allowing for lower maximum velocities, such as corners on many NASCAR tracks, or the famous "Carousel" corner on the Nürburgring race track. These can be taken at higher velocities than other corners with the same curvature, because the corners are sloped inwards: the shorter side of the bend is lower, allowing gravity to help the cars go around the bend.

While there are other factors, such as the difference between wet and dry roads and different road materials, the shape and banking angle of a track are fairly universal and not likely to change between tracks within one game. We therefore limit our model to just the shape (in particular, the radius of curvature of the road at a specific section) and the banking angle.

5.2 Model

Forces on a car We base our model on real life physics, so we first define what forces we will use. A breakdown of these forces (to be specific: the forces in the case of a velocity above the ideal velocity, as commonly found in racing games) can be found in Figure 5.1.

A car on a road experiences a gravitational force F_g pulling it down, equal to the constant of gravity g times the mass of the car m. This is counteracted by the normal force N, which can be seen as the road pushing back against the car. We are focused on when the car starts sliding sideways, in particular in corners, due to high velocity. This is explained by the centripetal force, $F_{centripetal}$, which is the force generated by objects rotating around a central point. In our case, the centripetal force is based on the mass of the car m, the radius of the curve r, and the velocity of the car v, as follows: $F_{centripetal} = \frac{mv^2}{r}$.

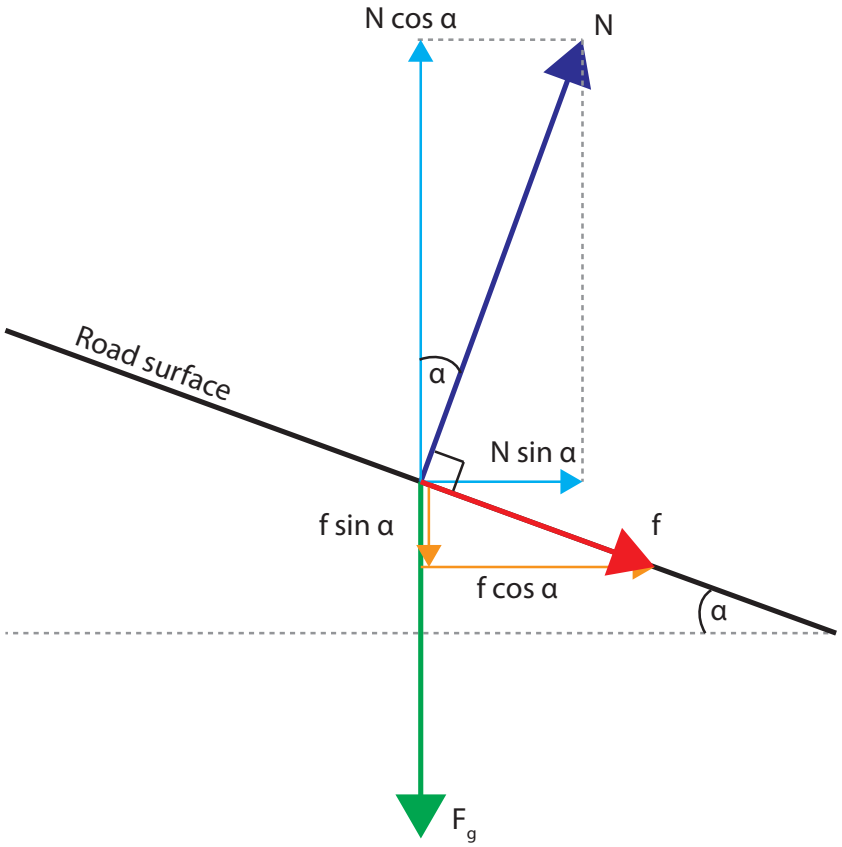

Figure 5.1: Forces on a car on a banked road. The normal force N is shown in dark blue, gravity F_g (or mg) in green, and friction force f in red. Horizontal and vertical components, as used in the maximum safe velocity v calculation, for the normal and friction forces shown in light blue and orange respectively.

A car driving through corners at normal velocities does have a centripetal force pushing it outward, without the car sliding off the road. On a flat road, this is mainly because the centripetal force is counteracted by the friction between the tires and the road. This form of friction, known as dry friction, is approximated using the Coulomb friction model: $f \leq \mu N$. The frictional force f is based on the coefficient of friction between the tires and the road, μ, as well as the normal force N. The car will start skidding towards the outside of a turn (the way the car is not turning) if the amount of friction force f is not enough to counteract the centripetal force. Because we are only interested in the situation where the friction force f is at its highest, we can further simplify this to $f = \mu N$.

The friction coefficient in our model depends mostly on game-dependent variables

such as the materials of the road and tires, weather conditions, the type of tire (rally cars have different tires than Formula 1 cars). While this may change during gameplay, these variables are constant in our implementation, so for our model we can consider it to be a constant value set by the game's designer. Had our implementation included tracks with wildly varying ground materials (dry asphalt, wet asphalt, dirt, snow) within the track, the model could have been expanded by storing a friction coefficient modifier corresponding to the ground material on each point along the track.

Banking roads On a road that is not perfectly flat, the banking angle of the road also plays a part: the normal force N now points at an angle. This will act as an force on the car: the car is pushed towards the lower edge of the road. If the road is banked towards the inside of a curve, the car is far less likely to start slipping to the outside, as gravity helps steer the car to the inside of the corner. However, if the road is banked towards the outside of the corner, it is far more likely to lose grip and slide off.

Formula for maximum safe velocity As can be seen in Figure 5.1, we have split the normal force N and the frictional force f into horizontal and vertical components, as this is required for the calculations. The horizontal components are $N \sin \alpha$ for the normal force, and $f \cos \alpha$ for the frictional force: these are the forces that prevent the car from sliding sideways (horizontally). At the maximum safe velocity before the car starts sliding sideways, the forces pushing the car towards the outer edge of the corner ($F_{centripetal}$) equal the net forces F_{net} that counteract this ($N \sin \alpha$ and $f \cos \alpha$). If the velocity goes above this maximum safe velocity, the centripetal force becomes too high and the car starts sliding.

Based on these forces, we can define a formula to calculate the maximum speed a car can drive without slipping. Using $F_{net} = N \sin \alpha + f \cos \alpha$, $N \cos \alpha = mg + f \sin \alpha$, and $f = \mu N$:

$$N \cos \alpha = mg + \mu N \sin \alpha$$

$$= N \cos \alpha - \mu N \sin \alpha = mg$$

$$= N(\cos \alpha - \mu \sin \alpha) = mg$$

$$= N = \frac{mg}{\cos \alpha - \mu \sin \alpha},$$

$$F_{net} = N \sin \alpha + f \cos \alpha$$

$$= F_{net} = N \sin \alpha + \mu N \cos \alpha$$

$$= F_{net} = N(\sin \alpha + \mu \cos \alpha)$$

$$= F_{net} = \frac{mg}{\cos \alpha - \mu \sin \alpha}(\sin \alpha + \mu \cos \alpha)$$

$$= F_{net} = \frac{\sin \alpha + \mu \cos \alpha}{\cos \alpha - \mu \sin \alpha} mg$$

$= F_{net} = \frac{tan\alpha+\mu}{1-\mu\tan\alpha}g.$

Considering this is equal to $F_{centripetal} = \frac{mv^2}{r}$, as explained on [25], we have now found the following formula: $v^2 = \frac{\tan\alpha+\mu}{1-(\mu\tan\alpha)}rg$, or: $v = \sqrt{\frac{\tan\alpha+\mu}{1-(\mu\tan\alpha)}rg}$. We will use this in our model.

Using the formula We assume a constant value for gravity g, as well as a set value for the friction coefficient μ. This can be based on the type of car and the amount of control desired by the game designers. So, to calculate the "maximum safe velocity" v, we then need radius of curvature r and banking angle α.

We expect to see an important correlation between the maximum safe velocity a car can drive at a certain point in the road, and the difficulty of the section surrounding this point. That is, we assume that points with a low value for v increases the difficulty of the section around it. However, we do not assume that points with a high value for v decrease the difficulty. For instance, if there is one very sharp corner at one point, flat and straight sections right before and after this point will not necessarily decrease the difficulty of driving through that sharp corner. As such, we are only interested in the points with the lowest values for v, or rather, the sections in which they are found. Similar to how we apply this to the data driven model (see Section 4.3), we define a section as a point and the k points in both directions from that point. Since we are only interested in the lowest values for v in a section, in our model, the lowest value v found for any of the points in a section will define the difficulty of the entire section.

5.3 Implementation

As with our data driven model (see Section 4.3), we start with the road as created in our testing environment. These procedurally generated roads (see Section 3.3) are based on splines. The points we calculate the maximum safe velocity v for are vectors along the same spline used to create the vertices of the road geometry. Using these vectors as points gives us some advantages. For starters, we do not need to do extra calculations to find points along the spline. The points are all in the middle of the road, giving a quite accurate representation. It also gives us the banking angle α we need at each point without extra calculations, as the banking angle is used to create the road geometry. During testing, we changed the amount of detail of the road geometry, generating more or less vertices along the same length of road. In case the road geometry was very detailed, and we felt the amount of points was too high, we could simply take every second or third point along the spline instead of every point.

Depending on implementation, the points along these splines may not be evenly spaced. In extreme cases, this may bias the classification of areas. For example, off

Figure 5.2: Markers floating above the road visualize the calculated maximum safe velocity v, ranging from high (green) to low (red).

road events on long straight sections may cover much more road than events in sharp corners. The spread of points should be corrected to prevent this from happening.

Calculating the maximum safe velocity To calculate v, we also need the radius of curvature r. As the points we use for our calculations accurately represent our road, we calculate this based on a circle through three sequential points along the curve. For a point p, r is defined as the radius of a circle through p, and the points immediately before and after p along the spline. Given a constant value for g and μ(based on the game implementation, as set by the game designer) we can now calculate v for all points along the spline. Simply inserting all values into the formula gives us v for all points. A visualization of this can be seen in Figure 5.2, where markers hover over the road at the point they represent. The markers are colored based on how high the (normalized) calculated values for v are: from high (green) to low (red).

Choosing a speed limit To define what values v are said to be low, and thus what sections are said to be difficult, we compare the maximum safe velocity v of a section to a threshold velocity t. If the value for v is below the threshold t, we define the section as difficult and expect the player to drive off the road, or crash. As with our data driven model, due to our implementation (which lacks walls on the sides of the road), we refer to these events as "off-road events".

Players are expected to go off-road at, or close to, points p with $v < t$. To test this hypothesis, as with our data driven model, we combine the road data (including

the calculated values of v for each point) with data of players driving along the roads. For each off-road event, the point along the spline closest to where the car left the road is marked as off-road. We define sections as a point p, and k points in both directions of the spline from p, resulting in $2k + 1$ sequential points. An off-road prediction for any point p with $v < t$ is marked as a true positive if an off-road event took place within the section with p as its center. If an off-road prediction is made, but no such off-road event is present within the section, it is marked as a false positive. Similarly, if an off-road event happened at a point q, but no sections containing q predict an off-road event happening, point q is marked as a false negative. All other points are said to be true negatives.

Setting the correct value of t A correct value for t is important to get valid results. A value t that is set too high will result in many false positives. That is, because too many values v are less than the high threshold t, we expect to see the driver go off-road far more often than will actually happen. If t is set too low, the opposite happens: very low values v, indicating sharp corners and banked roads, can still be above t and thus declared as not difficult. Setting the best value for t depends on many factors within the game world, including but not limited to the handling of the player's vehicle, the width of the road, and how well the car could brake. This emulates real life situations: a safe maximum velocity through a turn for a compact car is often much higher than that of a truck. Aside from practical reasons, it also depends on what the results will be used for. For some cases a potentially far higher amount of false positives may be worth it, as long as it ensures more or all true positives will also be found. Because of this, a correct value for t should be set by game designers based on their implementation, and possibly corrected after changes to the game's balance. It also means that, while we can at least try to explain what causes fluctuations in the output based on the value of t, we can not specify a best practice or best value.

5.4 Test results

5.4.1 Results

Unlike the data driven model, this model has very few variables that can be changed. The physics formula and its input are quite set in stone, leaving us with only the value for k to set manually. As the positions on the road are calculated in the exact same way in the data driven model as they are here, identical values for k produce identical road sections. Using the same value for both models means we are comparing the results of the models on identical road sections.

Based on the results we achieved with our data driven model, we decided to start with $k = 4$. This value was chosen as it produced good results for the data driven model, and might make comparing the two models easier.

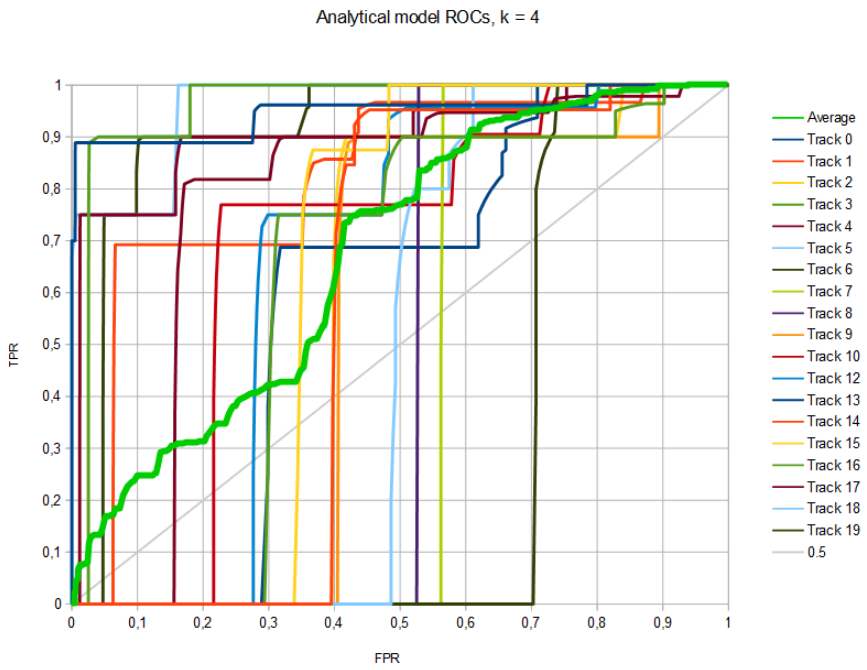

Figure 5.3: ROC curves for the results of the analytical model with $k = 4$

One run on several tracks Inserting the same data we used to find the results for our data driven model into the analytical model gives us the results displayed in Figure 5.3. The ROC curves are created by taking a manually set threshold value t increasing at a given interval, and calculating the true positive rate and false positive rate at each threshold. The average over all tracks is calculated by taking the mean of all these values for each value of t. The AUC values for these ROC curves are presented in Table 5.1.

The results for Track 11 are removed due to issues with a corner case in our analytical model implementation. We verified that this issue did not affect our testing or the results for other tracks, nor the results of the data driven model. As it only occurred once, and would have required a significant amount of time to fix, we decided against rewriting the implementation.

Repeated runs on one track These initial results show wildly varying AUC values. To help determine if the best and worst AUC results are dependent on the track or the player's performance on the track, we decided to do several laps on one track. We decided to use the track with the worst initial results, named Track 6, as it seemed to be an outlier. If the AUC score is mostly dependent on the track, it should be more likely to show similarly low results during following runs. The original results for this track, as well as the results for the additional 10 runs on the same track, are presented in Figure 5.4. The corresponding AUC values can be found in Table 5.2.

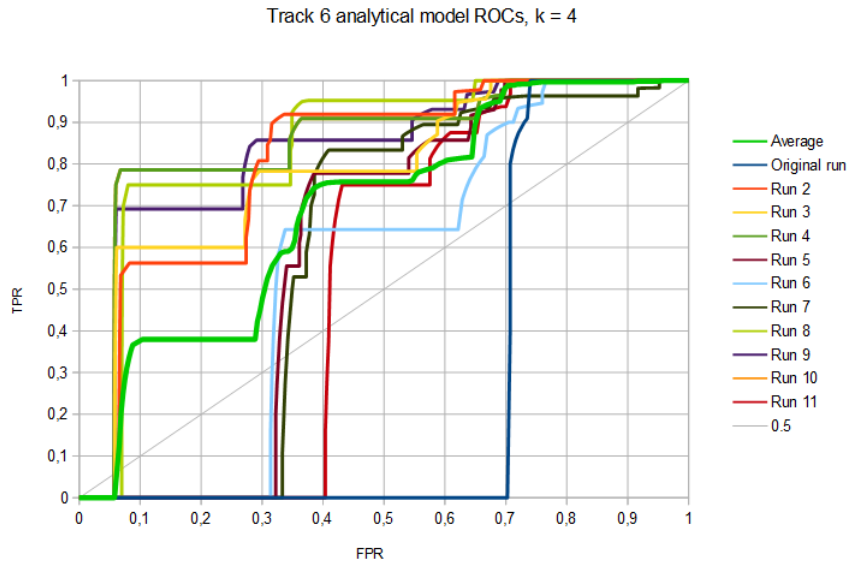

Figure 5.4: ROC curves for the results of the analytical model with $k = 4$, for multiple runs over Track 6

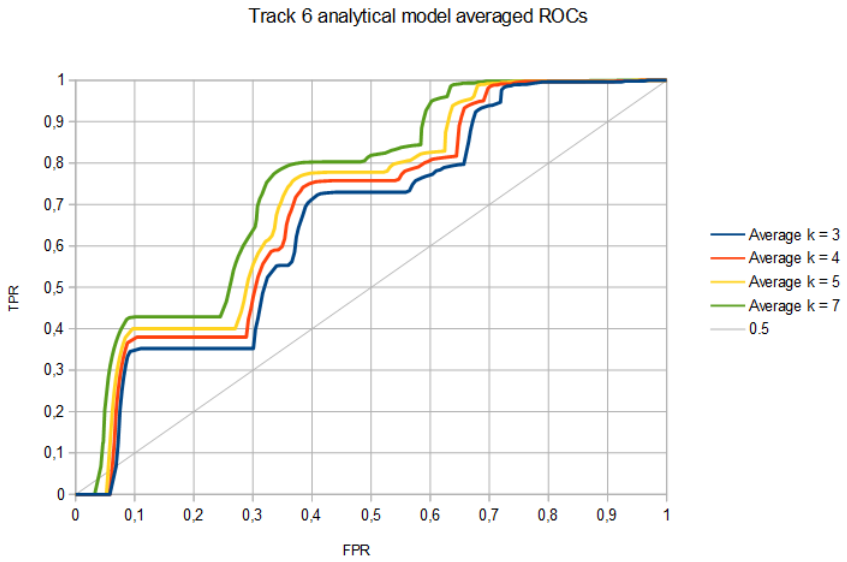

Figure 5.5: Averaged ROC curves for the results of the analytical model on multiple runs over Track 6, for varying values of k

Track	AUC	Track	AUC
0	0.949	10	0.685
1	0.577	12	0.656
2	0.636	13	0.589
3	0.616	14	0.821
4	0.776	15	0.558
5	0.487	16	0.959
6	**0.290**	17	0.906
7	0.437	18	**0.960**
8	0.473	19	0.914
9	0.544	Average	0.673

Table 5.1: Details for the results of the analytical model with $k = 4$, highest and lowest values in bold

Run #	AUC	Run #	AUC
1 (Original run)	**0.290**	7	0.592
2	0.809	8	0.846
3	0.783	9	0.829
4	0.850	10	**0.850**
5	0.598	11	0.534
6	0.554	Average	0.692

Table 5.2: Details for the results of the analytical model with $k = 4$, for multiple runs over Track 6, highest and lowest values in bold

Variables As mentioned, while there are not many variables to change, we did manually set the value for k. To see the effect of a varying value for k, we tried several options based on the averaged result for track 6. The results of this can be found in Figure 5.5.

5.4.2 Analysis

The average AUC scores in our test results were just below 0.7, which does not seem very high. However, we have to take into account that this model is designed to be applied to racing games. We can not expect to find every cause, or even the most likely cause for players to fail, especially not with just one model. There will always be outliers, such as objectively easy parts of tracks where players simply did not pay attention. As such, we can not expect to achieve the very high average AUC scores that are considered a minimum standard in fields that work with more objective measurements.

The first set of data, a single run over each of 19 different tracks, showed wildly

varying results. The second set of data, several runs over the worst performing track of the first set, show a similar pattern. In fact, the average AUC values for both sets are quite close together, with a difference of only 0.018 (about 0.02). Both sets of data show a big spread between best and worst results. From this we conclude that the model is very inaccurate.

Role of player performance As shown by the second data set, the player's performance plays a very big role: runs by the same player on the same road with fully identical settings created wildly differing results. When looking at the ROC curves for both sets, the lowest result seems to be somewhat of an outlier. But even when taking the second lowest result (0.534), the difference between that and the highest AUC value is still 0.316. This much variance ($\sigma^2 = 0.031$ for the full second set) on a track with otherwise identical settings shows that the player's performance is a very significant factor. This is important because it indicates that while the originally bad result for Track 6 was not repeated in further runs, the other tracks are also not guaranteed to always perform better.

Adjusting k The result of adjusting k is quite clear, as shown by Figure 5.5. As the size of the road section checked for an off-road event increases, the likelihood of finding true positives increases, while that of finding false negatives decreases. As such, the entire ROC slowly moves towards the top left corner, increasing the AUC value. However, at the same time, accuracy is lost. If k is set to roughly half the length of the road, the road section checked for off-road events is the same length as the road itself. That means that only one or two sections, surrounding positions in the very middle of the road, are checked. In almost all cases this will result in an ROC curve with an AUC of 1.0 if the player left the road at any point during the run, or an AUC of 0.0 if the player did not. As we assume players will always go off-road at least once on tracks if they are difficult enough (which in fact happened in all cases during our testing), clearly this will result in a very high average AUC. But as the actual accuracy is so low, it gives us no useful data. Therefore, we believe that in practice a relatively low value of k is actual beneficial for this model.

To conclude, we can say that longer it takes a player to travel the length of one road section (at maximum velocity), the less accurate the results will be. To keep the accuracy high enough for meaningful results, we suggest that k is chosen so that the area of road covered by $2k+1$ points (one section of road) does not exceed a certain distance. This distance can be defined as the distance traveled by the vehicle at its maximum velocity in, for example, 4 seconds. A similar lower bound is also required: a road section should cover at least a certain amount of road, similarly depending on the vehicle's maximum velocity, as otherwise a correctly identified difficult point causing an accident just a few tenths of a second later may create false positives.

How high k needs to be to meet these requirements depends on the racing game implementation (in particular, the maximum velocity), as well as the density of

points along the road. The exact time duration upon which either of these distances would be based is also dependent on the game's implementation and balance, as the car's handling heavily influences how much time players require to correct mistakes.

5.5 Derivative of maximum velocity

Looking closely at the data showed that off-road events often happened at or close to points with a value of v of 20 m/s or lower. Points along the track with a higher value for v usually had no off-road events nearby. As such, we tried setting t to 20 m/s (rather than calculating the ROC curve for a wide range of values for t), which indeed seemed to produce good results in many cases. Some outliers aside, there were many true positives, and very few false negatives or false positives. However, there were some exceptions: at times, off-road events happened around points with maximum speeds of 35 m/s or higher. As most points with such a high value for v had no off-road events nearby, our model would generally produce false negatives in these cases. Increasing t did not improve the results, as this would produce far more false positives than true positives.

Rapidly decreasing maximum safe velocities It quickly turned out that many cases of off-road events that we had not predicted followed a consistent pattern. The off-road events usually occurred at or near points with a value of v between 30 and 40 m/s, with points with a much higher value for v (frequently around 100-140 m/s) not far before it. This means that while the lowest value of v was relatively high, the difference between the highest and the lowest values was quite large in a relatively short amount of distance (or points along the track). In practice, this means that it requires the player to slow down considerably in a short amount of time, possibly in areas where they did not expect it.

These observations suggested that using the derivative of our calculated maximum velocity values was important. As our model was not suited to take this into account, and we could not find an easy way to add this functionality for quick testing, we did not apply it in our method. However, we do believe it could play a big role in making our model more accurate in the future.

5.6 Subjective analysis

The results of our analytical model are based on extremes. It only considers situations where the player lost control to the point the car went off the road, but not situations where the player barely managed to regain control. Likewise, only the sections with very low values for v are marked as difficult: anything slightly above the threshold t is marked as not difficult. This means that by looking just at the

objective results, a lot of possibly useful data is lost. By looking at how the rest of the calculated values compare to the player experience, we hope to provide some additional insight.

Looking at the data showed that areas that players considered not very difficult had relatively high values for v compared to more difficult sections. Average values commonly appeared on banked straight roads, or weak corners, while the highest values appeared on straight flat pieces of road. Hardly any off-road events occurred on these sections, and they were mostly considered not difficult by players.

Sharper corners around hills, banked the wrong way (the inside of the corner higher than the outside) were considered quite hard. Very sharp corners (usually an artifact of merging two calculated paths, halfway through the road) usually were considered the most difficult by players, and indeed had some of the lowest values for v as well as a big portion of the off-road events.

One specific example that players considered difficult, but which was not represented by the model, was roads over the top of relatively steep hills. At high speeds, this would sometimes "launch" the car. The wheels would lose contact with the road, meaning the player lost control of direction and sometimes go off the road. Especially on road sections that were not banked and mostly straight, the calculated value v would be quite high, meaning that such sections created some false negatives.

6 Comparison of models

In order to properly compare the data driven model to the analytical model, we can not just look at the results, but need to look at what it takes to achieve those results as well as what the results represent. In this section we will try to explain what these differences are and how they affect our interpretation of the results.

6.1 Comparison of the results

We found it very hard to draw conclusions from the results of our data driven model. The ROC curves we drew were based on the probability of certain sections containing an off-road event or not, as determined by the SVM. However, it includes no indication on what these probabilities are based on. Although SVMs are known for their black box nature, there are methods to provide some insight in how their classifications are made, such as the work by Barbella et al. [26]. However, as the SVM has so many factors as input $(3 * (2 * k + 1))$, it is hard to identify what is the most important contributing factor.

In addition to this, at first sight most of the results were less than stellar. Even with the best settings, the model barely reached an AUC of around 0.65, after optimization of the SVM on these specific cases. But player performance varies wildly, and even the most difficult situations may not always cause the player to fail, as long as the situation is not impossible to pass. As such, perfect results can not be expected, and expectations should not be set too high. Because of this, and given the lack of data to compare our results with, we can not easily conclude if these results are indeed as bad as they seem.

The results for the analytical model were not much higher, with an average AUC between 0.65 and 0.7 (although this can be increased, at a loss of accuracy: see Section 5.4). But while the data driven model had ROC curves based on estimates of a piece of software, the ROC curves for the analytical model were drawn based on a threshold on the maximum safe velocity we calculated. That data is far easier to immediately apply to the game's balance, as the results are based on fewer variables. Simply looking at the true and false positive rates for a certain maximum velocity threshold may give a game designer useful information.

6.2 Discussion and limitations of models

The two models have some key differences. Not only in the results, but also in the required input.

6.2.1 Discussion of differences

The added cost of training data The most obvious of these differences is the fact that the data driven model requires training, and thus, a far larger amount of data. The data driven model requires players to have driven over several tracks just to train the SVM. In addition to this, to prevent overfitting, these need to be many different tracks. This makes it rather unsuited for anything but a big, varied collection of tracks, accompanied by player data. While not a limitation of the model, it is a disadvantage in practice. Adding a way to reduce the amount of outliers before inserting the data into the SVM, and gathering results from multiple races over the same tracks, should greatly improve the results or at least reduce the amount of data required to achieve results.

There is an increasing amount of games allowing player generated levels that can be shared with (and played by) other players online. Such games would make an excellent fit for this method. However, if the track difficulty information is not required after but during development, this is likely not an option, unless there is a big amount of playtesters available. In fact, the entire model would be very difficult to use during development: any changes to the game balance (such as road width changes, friction changes, or changes to the car's speed, weight, acceleration, braking or handling) can influence what is or is not a difficult section. Thus, any change might require all training data to be discarded, as it is no longer accurate. Alternatively, it can be stored for later use, but additional data (representing the new variables) would still need to be gathered. As we have established that (for accurate results) we need relatively big amounts of data, this might cause issues for smaller development teams.

Repetition requirement The analytical model could be used with no data to train or even verify it. But in order to find a threshold velocity t below which accidents frequently occur, we need the data of at least one track and the record of one player doing one lap over this track. In practice this is not advised, as it is not guaranteed to always produce good results. As shown by our results in Section 5.4, results are very dependent on the player's performance. For more accurate results, it is advised to take the average of multiple runs instead. This means that the advantage of not requiring data for the training phase, as it is in the data driven model, is largely lost. However, while the data driven model requires different tracks to prevent overfitting the SVM, the analytical model does not. This will most likely make it easier to apply in real development.

Differences in input data As for the actual data to be inserted into the models, in our implementations the analytical model required only one more statistic about the road to be stored compared to the data driven model: the banking angle. For games that do not have a banking angle in their tracks, this does not apply, and the required input for the models can be identical. In fact, while we excluded it to keep the data driven model's input simple, the banking angle could be used and inserted into the SVM. This might help increase its accuracy, but also slightly increase the time required to train new SVM models. Regardless of the exact implementation, if set up correctly both models can be used side by side with very little or no change to the input.

Analysis of output The data driven model, as described in Section 6.1, has the drawback that its results are hard to analyze due to the black box aspect of SVM. A road, or section of road, may be qualified as difficult, or as not difficult. At most, the SVM can be configured to return a probability that a section is difficult. Analysis of why the section is difficult, however, is very hard if at all possible. This lack of insight into how a conclusion was made may prove to be a very big drawback for game designers, as it means they can not use this information to change the balance in the game.

Compared to this, the results of the analytical model are much easier to understand. In fact, it can be summarized in one sentence: very low threshold velocity means a section is very likely to be difficult. That is practical information that game designers could potentially use immediately during development, maybe even without doing test runs on tracks.

6.2.2 Limitations

Assumptions limiting application field Aside from the input, there is an important distinction between the models. The data driven model is based on pure data, while the analytical model incorporates assumptions on what causes difficulty. The maximum safe velocity is in reality just the maximum safe velocity before the car starts sliding sideways. However, as can be seen in rally driving, sliding sideways is in no way an absolute indication of difficulty. In certain categories of racing it can even be applied as a mechanic to gain the upper hand. Other physics, such as the car tipping rather than sliding, have not been included in the model. This might mean that for some types of race games, the data driven model is actually at a great advantage. It does not assume anything about what makes a track difficult. The SVM bases its information purely on player performance and the shape of the track rather than our interpretation of what that shape means for the player.

Limitations of both models Both models are only based on how the road itself is shaped, excluding many other potential factors. Our testing environment did

not include any obstacles, varying road widths or road surfaces, other players or buildings obstructing view around a corner. These can all easily be explained as potentially having some influence on the difficulty of the track, such as the road surface having a direct influence on the friction factor in our analytical model. One example of this is that even in our mostly wide open testing environment, we believe some of the unexpected crashes occurred due to the view being obstructed by hills. But this is not at all reflected in the results of either model. In fact, if any of these potential factors for difficulty occur on flat, mostly straight roads, our models will (possibly incorrectly) classify them as not at all difficult to drive on. Clearly, this is a limitation of both our models.

Possible solutions to this might be to expand the models (include road width and surface in the road data), combine our models with models which incorporate factors such as view obstruction and obstacles, or both.

Limitations of test environment While our models were designed to be potentially useful in a wide range of racing games, we have only managed to test them in one environment. Although we purposely limited our test environment to not include any obstacles to allow us to focus on the influence on difficulty of the road, this means we do not know how our models would perform in a full game environment. Because our models do not take obstacles, other players or varying road conditions into account, they might heavily influence the overall performance of our models.

It is currently unknown if the correctly identified difficult sections are equally difficult in such situations. For instance, players may drive around sharp corners more carefully if they expect other players to possibly crash into them, or if there can be unexpected obstacles on the road, or if the road may provide less grip due to sub-optimal road conditions. While this limitation applies to both our models, as our analytical model includes more assumptions about the environment (mostly based on our test settings), it is more likely to be influenced.

7 Conclusion

Analytical model We attempted to find a method with which we could identify the difficulty of a given track. We believe that the analytical model may have some use to pinpoint particularly tricky sections, by checking if low thresholds result in high amounts of positives. As explained in Section 6.2.2, our models are limited to the shape of the road. This means that not all difficult sections (such as those where hills obstructed the view of a moderately sharp corner) can be expected to be correctly classified, and indeed they were not.

However, many sections that players found difficult were correctly marked as difficult. Even for repeated runs by the same player over the same track, almost all of the most consistently difficult sections (in other words, the sections that were difficult even with prior knowledge of the track) were marked as having a low maximum speed, or a high difficulty, by this model. Additionally, we have not been able to find any comparable models, and consequently nothing to compare our results with. Because of this, we believe an AUC of nearly 0.7 is a very promising start.

As the analytical model does not have a training phase, it can be instantly applied to any track, making it easy to use. In practice, we believe this model could be used by racing game developers to quickly find some of the most tricky corners. These corners could then be adjusted to be less difficult, or the maximum velocity and acceleration of vehicles could be changed to make the player less likely to crash. Alternatively, the opposite could be done: if there are no sections marked as having a low maximum speed, sharper corners could be added, or the maximum velocity of vehicles could be increased.

Data driven model The data driven model did not meet our expectations of easily identifying difficult sections. We believe this is mostly due to our player data containing many data points that can not be linked to, or explained with, only the shape of the track as the player's performance varies too wildly. As such, our model is inadequate to draw definite conclusions for all our data.

Although the model itself correctly classifies over half the sections it is presented with, scoring nearly as well as the analytical model on paper, this does not assure it can be used in practice. The model gives only a binary classifier, or at most a probability, which stands only for if the player is expected go off road. This is often not enough to determine if a section is indeed difficult. The many variables make it harder to analyze how sections were, due to the SVMs black box aspect. This

means that we can not easily assert that the results are correct, or what they are based on.

Conclusion Our goal was to find a method with which the difficulty of new tracks could easily be identified. One of our main motivators for this was the increasing amount of procedurally generated content in games. We believed that racing games did not often contain procedurally generated racetracks because it is often hard to determine if a generated track is too easy, too hard or even impossible to drive on.

We designed two models to achieve this. First a data driven model, which takes the shape of the road and the performance of a player driving over it, and uses this to train a SVM implementation. Second an analytical model, which takes the shape of the road as well as the banking angle, and calculates a maximum velocity before the car would start sliding sideways according to the laws of physics.

Our analytical model has been shown to correctly identify many difficult sections. The variable on which this difficulty classification is based, the maximum safe velocity, is easily explained and easy to use. It can also easily be applied to any track, without the need of player data.

While our data driven model performed nearly as well on paper, it is much harder to draw conclusions from, due to the black box aspect of SVM. There is no clear variable that shows why a section is difficult, or what the most difficult point in a section is. In addition to this, it requires a lot of training data to properly function.

To conclude, we believe that while drawing useful information from the data driven model may be too complicated, the analytical model can indeed help identify the difficulty of tracks. However, both of the given difficulty models lack a high accuracy, as they are limited to basing their classification on the shape of the road. Additionally, the player's performance is a very influential factor in how well the models perform. Therefore, basing conclusions on only the output of the models is not advised.

7.1 Future work

Expanding the models We believe the models as presented here would form a good base to start from. The results we achieved with these models are far from ideal, and indicate that a lot of work can still be done to improve the models. We believe that for the data driven model replacing the input with less variables would be the best place to start. The calculated maximum safe velocity from the analytical model might in fact be an ideal first option.

For the analytical model, as mentioned in Section 5.5, we believe valuable information can be gained by looking at the derivative of the calculated maximum safe velocities along the track. Using the calculated maximum safe velocity as input for

machine learning, as suggested, might in fact help in identifying such patterns. We would also be interested in exploring other physics formulas, to see how they influence the results. In particular, the maximum velocity before a car starts tipping over due to centripetal force.

There are many other factors that help determine the difficulty in racing games, such as road width and surface variation, other cars, obstructed vision and obstacles on the road. We believe that incorporating these in the model, or combining our models with models based on these factors, is crucial in more accurately determining difficulty.

General difficulty models Developing, implementing and using difficulty models for games in general could also be considered important future work. As shown by Lopes and Bidarra [13] adaptivity and procedural content generation has seen some significant progress in recent years. However, as concluded by Hendrix et al. [10] most implementations focus on a few specific genres, or even just one genre, similar to how the models described here are specific to racing games.

Work such as that by Pedersen et al. [16] describe how the data gained from player experience models can be used to improve the experience of the player in a general way, but the model they implemented was specifically designed for a platformer. Moffett [17] went one step further, and split up his difficulty model into three layers; a generic layer, a refinement layer and an instantiation layer. The second and third layer are specific to a genre and a game respectively. While the generic layer applies to racing games in theory, not much information on how to proceed with implementing the second and third layer for game genres other than those described in Moffett's own work was given.

In our specific case, we were not able to find any work explaining to apply the knowledge gained by others in our environment. We have no way to conclude if models which focus heavily on factors that do not exist in racing games, such as equipment or enemies, would still work in racing games. As such, we believe that a lot of progress could be made by looking into ways to apply already existing methods to more specific genres. In particular, we are interested in work that details the steps one should go through to correctly apply a general model, such as that by Moffett [17], to a genre or game specific environment.

Automating difficulty assessment While the methods we described may help game designers to determine if tracks are indeed difficult to drive on, we do not yet have a way to automate this so it can be used for procedurally generated tracks. We believe this would be an interesting goal to aim for in the future.

Acknowledgments

I would like to thank my supervisors, Michael Wand and Marries van de Hoef, without whom I would not have gotten this far. They taught me much, and I am very grateful for their assistance.

Furthermore, I could not have done this without the support of my parents, who were always there for me whenever I needed them to be. And much love and thanks to Grace Ruppert, who supported me and tried her very best to help me in every way she could.

Finally, there have been many people who helped me in some small or not so small way, by giving insights or simply listening to what I had to say. They are too many to list here, and I don't want to place some people above others, but please know I am very grateful to all of you nonetheless.

And thank you, the reader, for reading all the way to the bottom here!

Bibliography

[1] Cardamone, Luigi. "Evolutionary learning and search-based content generation in computer games." (2012).

[2] Togelius, Julian, et al. "Search-based procedural content generation: A taxonomy and survey." Computational Intelligence and AI in Games, IEEE Transactions on 3.3 (2011): 172-186.

[3] Togelius, Julian, and Simon M. Lucas. "Evolving controllers for simulated car racing." Evolutionary Computation, 2005. The 2005 IEEE Congress on. Vol. 2. IEEE, 2005.

[4] Togelius, Julian, Renzo De Nardi, and Simon M. Lucas. "Making racing fun through player modeling and track evolution." (2006).

[5] Togelius, Julian, Renzo De Nardi, and Simon M. Lucas. "Towards automatic personalised content creation for racing games." Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on. IEEE, 2007.

[6] Shaker, Noor, Georgios N. Yannakakis, and Julian Togelius. "Towards Automatic Personalized Content Generation for Platform Games." AIIDE. 2010.

[7] Loiacono, Daniele, Luigi Cardamone, and Pier Luca Lanzi. "Automatic track generation for high-end racing games using evolutionary computation." Computational Intelligence and AI in Games, IEEE Transactions on 3.3 (2011): 245-259.

[8] Cardamone, Luigi, Daniele Loiacono, and Pier Luca Lanzi. "Interactive evolution for the procedural generation of tracks in a high-end racing game." Proceedings of the 13th annual conference on Genetic and evolutionary computation. ACM, 2011.

[9] Wang, Jiao Jian, and Olana Missura. "Racing tracks improvisation." Computational Intelligence and Games (CIG), 2014 IEEE Conference on. IEEE, 2014.

[10] Hendrikx, Mark, et al. "Procedural content generation for games: A survey." ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP) 9.1 (2013): 1.

[11] Lucas, Simon M., and Julian Togelius. "Point-to-point car racing: an initial study of evolution versus temporal difference learning." Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on. IEEE, 2007.

[12] Iosup, Alexandru. "POGGI: generating puzzle instances for online games on grid infrastructures." Concurrency and Computation: Practice and Experience 23.2 (2011): 158-171.

[13] Lopes, Ricardo, and Rafael Bidarra. "Adaptivity challenges in games and simulations: a survey." Computational Intelligence and AI in Games, IEEE Transactions on 3.2 (2011): 85-99.

[14] Togelius, Julian, and Jürgen Schmidhuber. "An experiment in automatic game design." Computational Intelligence and Games, 2008. CIG'08. IEEE Symposium On. IEEE, 2008.

[15] Yannakakis, Georgios N., and John Hallam. "Real-time game adaptation for optimizing player satisfaction." Computational Intelligence and AI in Games, IEEE Transactions on 1.2 (2009): 121-133.

[16] Pedersen, Christopher, Julian Togelius, and Georgios N. Yannakakis. "Modeling player experience for content creation." Computational Intelligence and AI in Games, IEEE Transactions on 2.1 (2010): 54-67.

[17] Moffett, Jeffrey Peter. Applying Causal Models to Dynamic Difficulty Adjustment in Video Games. Diss. Worcester Polytechnic Institute, 2010.

[18] Jennings-Teats, Martin, Gillian Smith, and Noah Wardrip-Fruin. "Polymorph: A model for dynamic level generation." Sixth Artificial Intelligence and Interactive Digital Entertainment Conference. 2010.

[19] Csikszentmihalyi, Mihaly. Flow. Springer Netherlands, 2014.

[20] R Core Team (2014). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/.

[21] David Meyer, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel and Friedrich Leisch (2014). e1071: Misc Functions of the Department of Statistics (e1071), TU Wien. R package version 1.6-4. http://CRAN.R-project.org/package=e1071.

[22] Sing T, Sander O, Beerenwinkel N and Lengauer T (2005). "ROCR: visualizing classifier performance in R." _Bioinformatics_, *21*(20), pp. 7881. http://rocr.bioinf.mpi-sb.mpg.de.

[23] Shaker, Noor and Togelius, Julian and Nelson, Mark J. Procedural Content Generation in Games: A Textbook and an Overview of Current Research. Springer, 2015.

[24] Koster, Raph. Theory of fun for game design. " O'Reilly Media, Inc.", 2013.

[25] Stanbrough, JL. "A Banked Turn With Friction." A Banked Turn With Friction. N.p., 6 Feb. 2006. Web. 28 June 2015. http://www.batesville.k12.in.us/physics/phynet/mechanics/circular%20motion/banked_with_friction.htm.

[26] Barbella, David, et al. "Understanding Support Vector Machine Classifications via a Recommender System-Like Approach." DMIN. 2009.