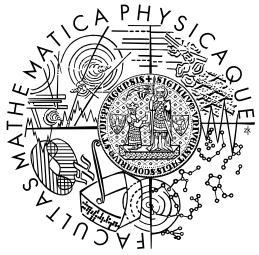

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Petr Šimůnek

Assisted race track procedural generation

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Martin Kahoun

Study programme: Computer Science - Software and Data Engineering

Prague 2022

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my family and friends for supporting me throughout my studies. I would also like to thank my supervisor for guiding me through the process, giving me the freedom to explore the solution when I wanted to, and for providing constructive feedback the whole time.

Title: Assisted race track procedural generation

Author: Bc. Petr Šimůnek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Martin Kahoun, Department of Software and Computer Science Education

Abstract: Racing games often take place on circuits with large surroundings. The goal of this thesis is to design and create a procedural generator for realistic road racing tracks surroundings. The track surroundings must feel populated and detailed during gameplay while also being performance-optimized for a smooth gaming experience. The generated environment needs to be consistent with the rules of road racing. Safety barriers need to be in places of high crash likelihood, and fans must be in relatively safe spectating spots. The generator will take the circuit shape, ideal racing line, and terrain as its inputs. Our method generates a 1×1 kilometer map in under a minute. Test players have rated our circuits as very realistic, detailed, and with more content than the previous hand-made ones. Our old circuits also took from dozens to hundreds of hours to create, considerably more than the new procedurally generated ones.

Keywords: procedural generation, runtime optimization, mixed-initiative, variable quality

Contents

1 Introduction **2**
1.1 The world of road racing . . . 3
1.2 The outline of the thesis . . . 6

2 Analysis **7**
2.1 Race track modelling . . . 7
2.2 Nature generation . . . 8
2.3 Procedural content generation in general . . . 9
2.4 High-performance rendering . . . 9

3 Outline of our method **11**
3.1 Simplified execution order . . . 12
3.2 Default parameters . . . 13
3.3 Implementation . . . 14

4 Our method in detail **15**
4.1 World analysis . . . 15
4.2 World generator initialization . . . 16
4.3 Generation based on normalized curves . . . 21
4.4 Translating generated data into the scene . . . 23
4.5 Nature generator . . . 32
4.6 Superbaker . . . 37
4.7 Results . . . 39

5 Evaluation **40**
5.1 User feedback . . . 40
5.2 Artist feedback . . . 40
5.3 Performance . . . 41
5.4 Future work . . . 41
5.5 Conclusion . . . 42

Bibliography **43**

Attachments **44**

A High-level overview of the classes in the attachment **45**

B Playable demo **47**

1. Introduction

In a typical racing game, the main gameplay happens on the racing track. However, although unimportant to the gameplay, the track surroundings are a big part of any racing game. Two parties interact with track surroundings in such a racing game: The players and the artists designing the tracks for the game.

There are two aspects by which players rate a racing track. First is the quality of the track layout. There are several subparameters players rate the track by, for example, opportunities to overtake, the combination of the corners, love or hate for chicanes, et cetera. Only a tiny change in millimeters can make the track much better or much worse in the eyes of the players.

The other aspect is the track surroundings. The track surroundings are essential to provide the atmosphere and sense of speed and make the circuit feel real. Therefore, we need to make the track surroundings as dense, detailed, and populated as possible. On the other hand, a high and stable framerate is crucial in the racing game to make correct split-second decisions while racing. Hence, the rendering process of the track surrounding needs to be as performant as possible.

The second party involved is the artists creating such tracks. When creating the track, we can once more split the process into two parts. First, the artist draws the layout of the circuit that he thinks could be good. Then he usually drives several laps and decides on changes to the circuit he wants to make and updates the circuit. The circuit is updated until the artist is happy. Sometimes, the artist can change the track design later based on received feedback from players. The track design usually takes several dozen minutes to a few hours of the artist's time.

The second step is creating track surroundings. Creating the track surroundings takes a tremendous amount of time in the magnitude of dozens to hundreds of hours if done properly, and matters very little to the gameplay itself. Also, it is tough to change the circuit after creating the surroundings since we need to delete the old one and create a new one.

One of the crucial aspects of the real circuit is the so-called ideal line. It is the optimal path around the circuit determined partially by physics laws like the conversation of energy and is greatly influenced by vehicle physics. The ideal line mostly travels through the corner from the outside to the apex, which is the point on the inside in the middle of the corner to the outside on the corner's exit. The problem is that we need to take the whole circuit into account. For example, if there is a straight coming up, it is beneficial to sacrifice some speed during the entry into the corner to have a better exit speed.

Furthermore, we want our ideal line to work in both directions since the race can take place in both directions. There are sometimes even several ideal lines through the corner in real life. Some ideal lines may be faster through a certain corner, but the entry into the following corner may be compromised. Some racing drivers would claim that the ideal line is something like art. For this reason, we opted to leave the responsibility for modeling the ideal line for the user. The user will model the ideal line with a curve in the road system and mark it as the ideal line.

We want to design a system that enables the artist to focus on creating the track layout as he did so far but automatically creates all the track surrounding with a reasonable amount of control. The goal of the thesis is to implement a system that takes the circuit, ideal line, and terrain as parameters and creates nice surroundings with high performance. The surroundings include safety features, fans, infrastructures like roads and paths, houses, and nature. The system needs to enable the artist to create the whole circuit with infrastructure, safety aspects, and others, and only nature is generated.

This work is based on an already implemented system for manually creating roads and placing objects for a racing game (Šimůnek [2020]). In this text, we refer to this system as *Road System* for simplicity. We aim to reuse this system and build on it when possible and extend this system when needed. Our system also contains completely new sub-systems, like a nature generator, designed these components so that they work with fully generated tracks and previously drawn tracks.

1.1 The world of road racing

Before we explain our method in detail, we would like to introduce the world of racing to the reader a little bit. Racing, and especially road racing modeled in this thesis, is a very particular kind of sport, where safety is almost non-existent, and the whole racing venue can be seen as anarchy by some. For example, it is normal that the fans of the sport can be seen inside the private gardens. For that reason, we want to warn the viewer that he will surely see several things he may consider dangerous or unrealistic. However, they are entirely normal during such events. We tried to point out these situations every time we came across them, but we may have seen too many of them in order to point out all of them.

When we model road racing tracks, we basically model a real-world environment with additional safety features and added fans. The safety features are added to the places with a high probability of crashes, and the race organizers select these spots based on their estimates and historical knowledge. The crashes usually happen when a motorbike or any vehicle loses control, it usually continues in its trajectory, which is not usually that much deviated from the road tangent.

For that reason, it should come as no surprise that the most dangerous spots are usually on the outside of the corners, most often the high-speed ones. On the other hand, the fans are located at spots considered safest. Therefore, the safest places are usually in the middle of the corners. Also, vehicles slow down at these parts, most action happens there, and usually biggest parts of the circuit are seen from such places. We can see an example of this in Figure 1.1.

Figure 1.1: Safety features, higlighted in green, are on the outside of the corner. Fans, higlighted in red, are standing on the inside of the corner. Source: `https://youtu.be/ovJ-CJPQjCg`, time: 0:30.

One thing to stress is that the fans of the sports are very passionate and manage to get into some situations that are considered a bit dangerous at best. The fans also very commonly enter the gardens of the houses next to the track; the landowners often allow this since it is a nice business opportunity, and they do not have too many options anyways in some cases. Such an example can be seen in Figure 1.2.

Figure 1.2: Fans standing inside the private property. Source: `https://youtu.be/ovJ-CJPQjCg`, time: 1:09.

One feature of the real world we want to stress is that biomes on each side of the road can differ. Of course, it is not always the case, but sometimes it occurs and gives the place a unique atmosphere. We can see an example of this in Figure 1.3.

Figure 1.3: Different biomes on each side of the road. Source: `https://youtu.be/ovJ-CJPQjCg`, time: 1:30.

Many things create the atmosphere of the track, one of them is the far background of the track. Such an example can be seen in Figure 1.4.

Figure 1.4: Track background can be a big part of the atmosphere of the track. It is highlighted inside the red ellipse in this figure. We can see some smaller hill with some vegetation in the distance. Source: `https://youtu.be/ovJ-CJPQjCg`, time: 0:28.

We would like to end the analysis with the conclusion that the racing environment is not always logical. It involves many passionate fans and politics, and it is sometimes visible on the racing tracks surroundings. On the other hand, the games are mostly played by players that want to race and play for fun. For that reason, it is much more critical for the tracing game for the surroundings to feel dense, live, plausible, and authentic; they do not have to be really realistic in all aspects; the impression is much more important by far.

1.2 The outline of the thesis

The rest of this thesis is organized as follows: In Chapter 1, we introduce the topic of our thesis and introduce the problems at hand. We then analyze the real world in detail and discuss the relevant work and technologies for our research in Chapter 2. In Chapter 3, we outline the main ideas behind our method and introduce the basic design decisions, and we discuss them in detail in Chapter 4. We discuss if the goals of the thesis were achieved in Chapter 5.

2. Analysis

On the following pages we disseminate our topic from a theoretical and practical standpoint. We present work related to race track modelling itself, vegetation generation and procedural modelling in general.

We also mention several related projects both freely and commercially available as well as the platform of our choice for implementing our work - the *Unity Engine*[1].

We selected this engine because we believe it is the best fit for our development. We occasionally try to bypass some of the functionalities of the engine in order to improve the performance. *Unity Engine* dictates many of the architectural decisions of our system; this is best demonstrated when talking about the terrain.

2.1 Race track modelling

Modelling a race track itself can be done either by hand, procedurally, or using a hybrid approach. An example of the former is our previous work the *Road System* (Šimůnek [2020]). The *Road System* is a tool for drawing roads and side objects in *Unity Engine*. In this thesis, we further extend this work and implement additional features. The tool works with placing prefabs, generating meshes, and spawning animated prefabs. The workflow is such that we create segments of the circuit separately in the *Unity Editor*. There are three possible modes how to interact with the tool:

- Runtime mode — when we play the game.
- Preview mode — mode when we model the circuit in the editor, and we need a real-time preview.
- Bake mode — mode, where we bake the final high-resolution result.

The tool is focused on user-friendliness, implementing simple yet powerful UI, it is also very flexible and gives excellent control of the drawn result. For example, the user can model changing road's width, curves of general shape, crossroads of any shape with changing asphalts, et cetera.

Artists first create the racing track and ideal line, and then they manually populate the surroundings with safety nets, fans, crossroads, et cetera. One of the generated outputs of our system is the control points for the *Road System*.

Similar project to the previous work would be commercially available *Race Track Generator* (MasterPixel3D [2018]) asset from the *Unity asset store*[2]. The first difference is that *Race Track Generator* generates only racing tracks used by top levels of motorsport. We focus on *road racing* circuits located in everyday locations like villages, forests, and fields; the *Race Track Generator* does not model these biomes. This solution also does not support changing the width of the circuit. We also target a denser environment.

[1] `https://unity.com/`

[2] `https://assetstore.unity.com/`

There is also a project of the same name available on the GitHub, the *Race Track Generator* (Hudson [2019]), it is an exciting work, which generates Voronoi cells (Aurenhammer [1991]) and then the racing track is based on the diagram cells. The results look exciting and pleasant; their shape can be edited well to the artists' liking. However, the first problem is that we do not have precise control over the width of the circuit, and the tool does not generate the track surroundings, landscape, or infrastructure, nor is there an option for drawing objects other than the road. Finally, our problem is not the artist not having the inspiration to create a fantastic circuit; our problem is to create the surroundings.

Similar to that one is the *Procedural Racetrack Generation* (Maciel [2020]). You cannot edit the road width, and the tool does not generate the track surroundings or infrastructure. The downsides are nearly the same as with the previous solution.

We want to mention also works of Daniele Loiacono (Loiacono et al. [2011]) and Ulrich Göhner (Göhner [2020]). The former generates the shape of the track and simple track surroundings. However, as stated in the introduction and the previous paragraphs, we want to give the user maximum control over the result.

Also, our opinions about what makes the racing track great differ. The authors of the paper state that it is essential to have diversity in the racing track to make it feel good. We believe it is far more critical for each track to have its distinct feel, achievable only by the human being present during the track design process.

Otherwise, the work is a similar case to *Race Track Generator* (Hudson [2019]) and *Procedural Racetrack Generation* (Maciel [2020]); you cannot edit the road width, and the tool does not generate the track surroundings or infrastructure.

The latter is the *Procedural race track generation for domain randomization* (Göhner [2020]), it tackles the problem of generating tracks for training the AI. As stated in the analysis, a slight change in the racing circuit can change its perception tremendously from the racing driver's perspective. Also, our thesis focuses on creating authentic surroundings for the track. The track itself is the user's domain, and the AI is not the focus of this thesis.

2.2 Nature generation

The *Vegetation Studio Pro* (Technologies [2018a]) is a package for the high-performance placement and rendering of vegetation available on the Unity asset store. After our analysis, we concluded that it has many great features and problems. It is great in terms of rule-based generation, and it is possible to define so-called vegetation masks to exclude vegetation from certain places. It is also possible to create different biomes to make the parts of the track feel special.

On the other hand, different density levels could be done only by introducing new biomes. If two biomes used the same prefab, it could not draw those at once. Also, it contains many bugs, and even a minor version sometimes breaks the package. It depends on four Unity packages; some are experimental and still evolving. Vegetation masks have poor precision for our purpose. Compatibility with custom shaders is complex. It generates billboards, but there is no option to have custom ones. We used this asset for nature generation and rendering before we implemented our own module for that.

Although *Vegetation Studio Pro* is an exciting asset and powerful in many ways, its best use for this thesis is probably an example of how not to design software. It also shows what design decisions should be avoided if we want to write stable code which is portable between *Unity Engine* versions without significant problems.

2.3 Procedural content generation in general

A Survey on the Procedural Generation of Virtual Worlds (Freiknecht and Effelsberg [2017]) is the most relevant article for our work by far. There were many relevant subjects such as terrain texturing workflow, spline-based editor, and Voronoi diagrams. This article also discusses the performance tradeoffs which came helpful when optimizing performance or preparing prefabs for the demo game.

The article suggests controlling the density of the vegetation with a grayscale map. Our approach is slightly different where two automatically generated grayscale maps are used. The first map describes the importance to the gameplay, and the second represents the depth of the forest. For tree placing, we used *Poisson disk distribution* (Särndal et al. [1992]), also described in the article because of its simplicity and speed.

The road generation was discussed as well. We believe that the *Road System* (Šimůnek [2020]) produces more optimized results and is better for creating side objects and other objects that follow a curve. The *Road System* can also model the racing circuit more precisely, including the ideal line and road width changing.

2.4 High-performance rendering

One technology that is widely used in computer graphics is GPU instancing. The GPU instancing plays a crucial role in our system since there aren't that many different features around the track, but there are high counts of them. Unity supports GPU instancing, but there is an asset called *GPU Instancer* (Technologies [2018b]) that pushes this functionality to the next level. Prefabs only need to be assigned to it in the UI, creating so-called prototypes. It bolsters frustum culling and occlusion culling features and has a simple yet powerful API. The crucial API for our usage is initializing it with a prototype and an array of transform matrices. It can be used without coding, simply by leaving the prefab in the scene. It is well documented and has great portability between Unity versions. After research we have done on the Unity Asset store, it is, in our opinion, the most powerful Unity asset available. It works with custom shaders very well, which will enable us to do some optimizations down the line.

The traditional approach in working with Unity scenes is that *GameObjects* are placed and spawned in the scene. These *GameObjects* can contain such components as *Transform*, *MeshRenderer*, *MeshFilter*, *Collider*s, and custom scripts. When *GPU Instancer* is initialized in the *Start()* method, it scans the whole scene and disables all *MeshRenderer*s, so the rendering is done only once by *GPU Instancer* and not by the *Unity Engine*. We can initialize *GPU Instancer* with a *Matrix4x4* array of the *GameObject*s we want to render while disabling

all *MeshRenderer* components in Editor. This reduces the time *Start()* method takes, but we can do better. We can remove the original *GameObject*s, resulting in much faster load times and reduced scene size. This approach is called *NoGameObject* workflow. One thing we must handle is the *Collider*s; we need to leave them in the scene in order for collisions to work. Not all *GameObject*s have *Collider*s, and we can also reduce their count by removing unreachable ones.

The *GPU Instancer - Crowd Animations* (Technologies [2019]) is an extension of *GPU Instancer* for skinned meshes, for example, animated fans.

3. Outline of our method

In this chapter, we give the reader a high-level overview of the method. We describe the whole algorithm in detail in Chapter 4. We also describe the execution order and how some of the default parameters were decided, and we also introduce implementation requirements for the interaction modes.

The method is split into several logical components, with each having one responsibility. The first part is an improved road and side object drawing system based on the old *Road System* (Šimůnek [2020]). This tool is responsible for translating curves drawn in the Unity Editor into roads, fans, safety packages, et cetera. One of the goals of the generation process is to generate those curves with metadata that the user would otherwise manually create.

We start by splitting the world into cells; each cell has its biome, like a city, forest, or field. Next, we generate these cells such that edges follow the road. We achieve this using adaptive curve sampling and then dissolving edges between cells that are too close. Finally, we use the Voronoi diagram (Aurenhammer [1991]) for generating these cells.

To provide sufficient information for further steps of the generation process, we store information such as distance to the racetrack, biome, et cetera., for each point inside the world generating structure. Based on this information, we will control the level of detail, decide if prefabs should be spawned, and much more. We provide such features by creating a 2D structure that caches this information, and we then get it in a fast lookup. We later discuss the implementation of these features in detail.

As part of this world generation structure, we model the whole graph, including edges, Voronoi centroids, and vertices in the edge graph, all cross-referenced, to navigate between graph features easily. In addition, these contain other world space functions like finding the minimal distance from the track along the edge or getting the distance from the edge. This world-generating structure can also generate critical features of the world, like minor roads or dirt paths. The main goal of this component is to provide a good description of the generated world. This allows us to write simple logic as:

- For all curves: draw houses.
- For all Voronoi: texture terrain under it.
- For the main curve: draw safety features.
- For all city Voronoi, draw bushes in the Voronoi inner hull, without edges with an asphalt road.

This ensures that the high-level code is simple, resulting in better customizability if any changes are needed.

The next component we need is a translator from simple instructions to the road and side object drawers. We do this in two parts, first is an extended API for the *Road System*, including functions as draw path from point A to B. The second translation component is the so-called normalization structure, which allows us to have fast and precise control over the *Beziér curves* (de Casteljau

[1959]) used by the *Road System*. It is responsible for the lower level generation logic as deciding where is dangerous zone and safety features should be spawned or where the spectator area is. It is also responsible for final details like creating an entrance for each house.

One more important component of our scene is the terrain. The terrain is implemented in a specific way in the *Unity Engine*. It is controlled by textures: a square heightmap with a resolution of $2^x + 1$ for controlling the height of the terrain and a square splat map with a resolution of 2^x for controlling the texture of the terrain. The heightmap is a 2D array with the size of the control texture with float values between 0 and 1. The splat map is a 3D array with a size of the control texture and number of terrain splats = number of textures. The sum of splat layers has to be exactly one at any point. The resolution of control textures never matches. Because how the terrain is a separate asset, every operation is destructive. There is not any simple API to access the terrain component. All API needs to work with the arrays directly. Thus, we implemented another level of abstraction called *TerrainWrapper*, which provides such information in simple world space API.

Next, we generate nature; this nature generator is focused on performance, and it is the best demonstration of variable quality based on gameplay importance. We need this generator to work with both manually created and generated maps, so we base it on the terrain splat mapping. In other words, based on the texturing of the terrain. We generate different types of forests and flowers to make the atmosphere of each map more special.

The final process is named *superbake*. This process converts all *GPU Instancer prototypes* into *NoGameObject* workflow and culls unreachable colliders. This means that most objects are only defined by their matrices, which will be directly uploaded to the GPU, not represented in the scene, thus taking the responsibility off the *Unity Engine*, saving the performance and resulting in minimal build size and load times while giving us excellent control over the rendering process.

3.1 Simplified execution order

Map generation follows these steps:

- Interpret user input into a *NavigationSupport* and *GenerationGrid* to obtain a normalized description of the drawn circuit.
- Generate centroids of Voronoi cells and create the world generating definition and prepare data for all its functions.
- Translate generated data into input data for road drawing system using simple core execution logic.
- Paint terrain splat map.
- Draw the roads and side objects.
- Commit terrain changes.
- Generate nature using nature generator.

- Superbake the scene into *NoGameObject* workflow.
- Clear all variables to ensure that they do not get realized by Unity and do not increase the final build size.

3.2 Default parameters

Our system will have many parameters exposed to the user. Therefore, one of our goals was to find default parameters that provide satisfactory results. In the following paragraphs, we will explain the motivation behind some of the values.

First, we would like to bound the size of the circuit. Each track is a loop with a length between 500 m and 8 km. The track cannot fit in square 100×100 meters but can fit in square 4096×4096 meters.

We have made the following assumptions about the gameplay: Game mechanics force players to spend time on the racing track, and they cannot spend more than 3 seconds outside of the track. The maximum speed any vehicle can move is $300\,\mathrm{km/h}$ ($\approx 80\,\mathrm{m/s}$). From empirical research, we can state the following about players who leave the track unintentionally:

- They either leave the track at the sharp turn. In that case, there is a barrier[1]. However, the distance traveled outside the track, in this case, is only several meters.
- In the second case, they leave the track on the straight or mild corner. Then the angle between the vehicle direction and racing track tangent is only an insignificant angle. We estimate this angle to be $10°$.

The critical thing to note is that most crashes often do not happen at the maximum speed. The average high-speed crash happens at less than $200\,\mathrm{km/h} \approx 55\,\mathrm{m/s}$. Using this assumption, we can calculate the maximum reachable distance from the track to be:

$$\sin(\alpha) \cdot v \cdot t = s,$$

$$\sin(10°) \cdot 55 \cdot 3 \approx 29\,\mathrm{m}.$$

Most of the parameters in our system were decided empirically end fine-tuned based on the feedback of the thesis supervisor and test players. Most values are based on the maximum reachable distance from the track, which we have estimated now.

[1] If the map is generated, there should be a barrier placed by our generator or the user otherwise.

3.3 Implementation

In Chapter 2.1 we discussed the interaction modes with the *Road System*, and this design was so good that we used it and extended it. In our system, we imposed the following constraints described in Table 3.1. For clarity reasons: *NoGO* = *NoGameObject*.

	Runtime	Preview	Bake
Interactive	no	yes	no
Working with road segment count	all	single	all
CPU time	0.05 ms	200 ms	1 minute
GPU optimization	high	low	high
Max allocations	0	dozen	no limit
Quality	high	low	high
Memory leaks	forbidden	forbidden	undesirable
Stability	crucial	medium	high
NoGO work-flow needed	yes	no	yes

Table 3.1: Modes requirements

4. Our method in detail

In this chapter, we expand on the algorithm of generating the race track surroundings already introduced in Chapter 3 and describe it in detail.

4.1 World analysis

The scene generation process starts with the user clicking on the Bake button. After that, *RoadGroup* is found, and the main bake function loop is called, which executes all the steps described in Section 3.1. This text assumes that all generating features are turned on, in case they are not, most features are computed anyway but not written into the scene since some features from generating API are still essential for some computations, especially for control over the level of details; for example, the distance function is needed when generating the nature.

We start the generating process itself by the primary generating function. First, we ensure the terrain and all needed components mentioned in the introduction, like *NatureManager*, are present and get references to them. We also load all definitions of side objects, prefabs, material, and *Road System* settings.

After this, we start to analyze what the circuit looks like. We start this analysis process by building the *NavigationSupport*, which represents the area of the circuit. This structure also handles support for AI. We start by regularly sampling each main curve of each road segment; in each sample, we create *RoadNavigationFrame*, which resembles position in the world space, road tangent, normal, and width.

Next, we need to compute the position of the ideal line in each navigation frame; we do that by regularly sampling the curve representing the ideal line; then, we have two lists of samples that follow a roughly similar path. We find the closest point in the ideal line to each navigation frame by marching in both lists simultaneously. Next, based on the ideal line, we can calculate other values for the AI. These values are based on the vehicles' physics; hence they are not part of the thesis. We recommend starting with *normalized angles* in both directions and implementing a geometric dropout. A *normalized angle* is defined as the angle between two neighboring frames, divided by their distance. The final step is to instantiate transparent prefabs to help less experienced players around the circuit. We can see the ideal line visualized in Figure 4.1.

Figure 4.1: The ideal line is visualized by the green arrows. The big red arrows are pointing toward the ideal line.

4.2 World generator initialization

Many of the subsequent generation steps need a fast distance from the track calculation. To facilitate it, as a pre-processing step, we calculate a distance function for selected points and store the data in a 2D array similar to a texture. We start by allocating a 2D grid with the size of the terrain located in the terrain's local space. The distance between points in the grid is by default 1 meter for simplicity. Then, we start by iterating through the whole navigation grid and triangulating its area. We get all points inside the navigation grid area (= contained inside at least one of the triangles).

Finally, we assign all points inside the track the value of zero and set them as dirty; all other points have the value of infinity. We then propagate the distances. For all dirty points, we check all neighboring points under a given distance; if the distance to the current point plus the distance between these two points is smaller than the current value in the updated point, we update the value and mark the point as dirty. We continue as long as there are dirty points. The finishing step is to construct a function to access this grid from the world space. We achieve this by transforming the world space point into a terrain space point and finding the point in the grid, which is done by simple rounding. This approach proved to be good enough by far for our usage. In a more advanced version, interpolation could enhance this approach while precisely computing selected points, but this is a topic for further research if needed. The presented approach is straightforward and works well. We can see a visualization of the distance function in Figure 4.2.

Figure 4.2: Distance texture visualization. The black areas are areas directly inside the circuit. White areas are those places very far from the circuit. The brighter the given spot is, the more distant from the track it is. Please note that due to how the colors are displayed on your device or physical copy, some spots close to the track may appear black as well. Please ignore the text at the bottom of the figure.

Now comes the critical part of the algorithm; we generate the Voronoi diagram, which divides the world into cells. It is good to remark that the world generation method is independent on the cell creation scheme. It could be a square grid, triangle grid, Voronoi diagram, et cetera. We chose the Voronoi diagram because it can produce many variable shapes, each cell can be different, and the shape of the track can be complex; the Voronoi diagram can fit that well, the result produced by the Voronoi diagram looks the most natural for our usage.

We start by generating centroids; there are two types of centroids, the first follows the road to ensure that biomes change along the track, and the second type is randomly generated centroids. We want the first type to follow the racing track so the Voronoi cell edges follow it too and split the racing track in the middle to have the possibility of having different biomes on the opposite sides of the racing track.

We use adaptive sampling and generate centroids between the samples to have this edge split as close to the middle of the road. We add a centroid to each side at the beginning to avoid segment ownership problems discussed later.

In the second stage, we generate centroids randomly; we only ensure that each generated centroid is further from the track than the distance parameter from adaptive sampling divided by two. This ensures that our edges in the middle fulfill desired properties. The example of the resulting centroids can be visible in Figure 4.3.

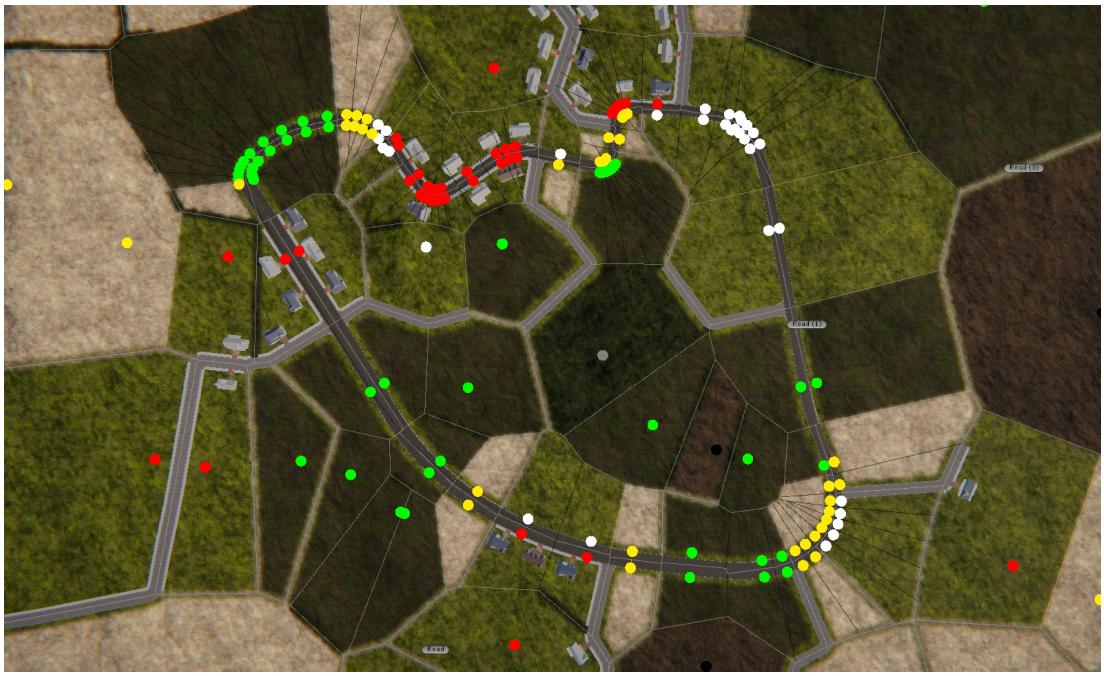

Figure 4.3: Centroids visualization, each dot is a centroid. The color scheme is following: red = city, white = meadow, yellow = field, green = forest, black = clearing, grey = small dense forest.

After generating desired centroids, we compute the Voronoi graph. We opted to use the *csDelaunay*[1] library for this task; the critical remark is that if any other library were used, it would impact only minor parts of this generating structure and not the rest of the system. We parse returned data structure and transform it into classes, representing cells, edges between them, and vertices between edges while cross-referencing them. These classes are based on Unity vectors and located in world space to hide implementation details and ensure easy version portability. This library does not produce edges at the map edges and map, and in the corners, we create those, and thanks to this, we can easily mark what edges are border edges.

The next step after the world subdivision into cells is biome assignment to each cell. To create an interesting and varied environment, we use a function taking world position and scale as input and map the intervals from the result range onto specific biomes. We use *Perlin noise* (Perlin [1985]) for this role as it produces nice results and fulfils the criteria, and is simple to explain to the user of the system.

Lastly, we can force all cells to one specified biome or override specific cells with a selected biome. It is achieved with a simple yet powerful workflow based on drawing circle overrides and editing them (move and scale), as demonstrated in Figure 4.4.

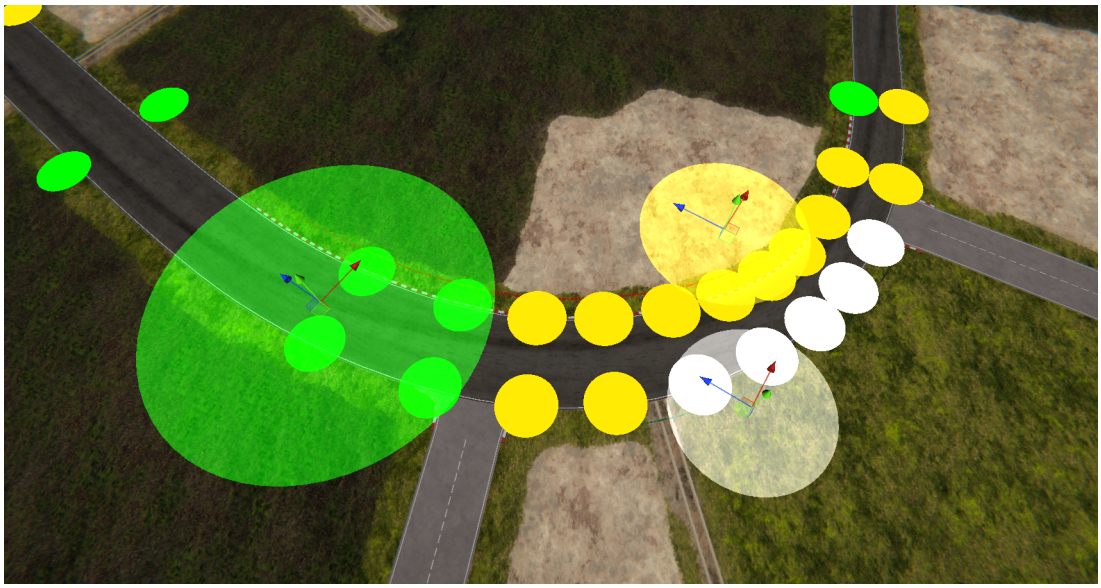

Figure 4.4: Biome override example. The color scheme is following: white = meadow, yellow = field, green = forest. The full circles are centroids; the transparent circles are the overrides.

[1] *csDelaunay* `https://github.com/PouletFrit/csDelaunay` is a C# version of the *as3delaunay* libraly `http://nodename.github.io/as3delaunay/`

After all centroids and edges have been initialized, and the graph is cross-linked, we come back to solve one problem caused by the adaptive sampling while generating the centroids. The problem is that the cells can be too tightly packed in sharp corners, as seen in Figure 4.3, creating cells in the shape of long noodles that do not look good. We solve this issue by so-called edge dissolving. Edge dissolving is implemented as a bool flag if the edge is dissolved or not.

This approach is intuitive and straightforward yet does the job really well. We merge all cells with centroids closer than the maximum distance parameter from adaptive sampling divided by two on each side of the road with the same biome for the best result. The critical thing to stress out is that we never merge centroids on the opposite side of the road; that would break the edge division in the middle of the road.

The last thing we need to determine is where the side roads and dirt roads are connecting the circuit to the outside world. We use the diagram edges as potential paths for our roads and paths. Site road creation is a two-step process. First, we iterate through all Voronoi edge vertices, selecting those in the middle of the road. Then we flip a coin, and depending on the result, we either start creating a road or continue. Probabilities are altered based on the biome at the place.

Next, we try to travel from the vertex through the world until we are able to. The path ends when we are at the edge of the world or there are no suitable edges. We pick the next vertex by selecting the edge with the highest gradient. Edge gradient is defined as the difference between the track distance of the edge vertices.

It is essential to state that this algorithm was selected because it feels the most natural when racing around the track. Paths are created similarly, but they are cheaper to build and maintain in real life, so they meander more. For this reason, they flip a coin when selecting a path between an optimal road path and a random edge; we only select those with sufficient gradients to stop the path from going back to the main road. Paths are created from the main road and side roads; for this reason, they are generated in the second iteration. Finally, we assign each edge width based on the road/path/dissolved/another status. We can see generated roads and paths in Figure 4.5.

Figure 4.5: Generated roads in red and generated paths in yellow.

4.3 Generation based on normalized curves

With the Voronoi diagram initialized, we create the *GenerationGrid* structure, which resembles the edges of the curves. We initialize it by regularly sampling all main curves from all *Road System* segments first as one and then each other road curve individually. This is done to take into account that the circuit is a loop. Each sample is represented by *GeneratingFrame*, which provides API to tell us if a specific type of side object is present. Several examples of side objects are normal curbs, highlighted curbs, safety nets, fans, safety packages, or houses. Using the updated *Road System*, we draw these side objects by finding all intervals and drawing corresponding side objects from the beginning to the end. The last and the most critical puzzle piece is implementing the function, returning the value if the side object of a given type is present. Each side object is unique. Hence we implement it with different logic. We precompute this logic when creating the *GeneratingFrames* in several steps. Each logic for each side object is distinct, and we will not describe all steps here since it is not that interesting and would take long.

We start by caching in each *GeneratingFrame* if it is relatively close to the edge. Relative distance from the edge is defined as the distance from the edge divided by edge width. We get this information from generating diagram API. The second step is caching the biome per *GeneratingFrame* since some side objects differ depending on the surrounding biome and calculating the *normalized angle*. Next, we assign safety packages to the outside of the corners based on the *normalized angle*. The *normalized angle* is above 0.9 in our case for safety packages. Fans are at areas with a *normalized angle* below 0.1 (corner insides and straights). Curbs are at areas with *normalized angle* below -0.2 (corners insides).

We then do a post-processing process, where we dilate the safety features and erode fans to have a safety margin. Next, we generate houses; we start by checking if the biome is a city, then if the center of the house is located far enough from already generated houses centers. Lastly, we check all corners of the house to see if there is enough distance between them and the nearest edge bound. The edge bound distance is defined as the distance from the edge minus edge width.

Finally, we check if the distance from the track is between the minimum and maximum allowed value; these two tests first make sure that the house is not too close to the road to interact with the gameplay physically, and second, cull houses that are too far away to save the performance. If the distance is smaller than the maximum allowed for houses, we generate bushes as well. Finally, we make a simple entrance to each house. We know where houses are, so we make bush cutouts, cut out in the sidewalk, align ends and spawn a low sidewalk that overextends a little bit. Such house entrance is demonstrated in Figure 4.6.

Figure 4.6: House entrance with bush cutout and lowered sidewalk.

The last *GeneratinGrid* step is removing fans, sidewalks, and other non-safety features when a path is present, so we disable them when the relative distance from the edge is below 1. We can see such an edge cutout in Figure 4.7. This structure was initially designed for the main curve but trivially extends to other lines; we do not spawn safety features and fans there because there is no need for any protection.

Figure 4.7: Cutout in fans side object using relative distance from the edge API. The orange fence is not a safety feature in this case.

4.4 Translating generated data into the scene

So far, we have done nothing in our scene; all was done only in raw C# classes; now, we need to convert it with *GameObject*s, *Collider*s, *Mesh*es, and other *Component*s. We start by initializing some of the components we will discuss later in detail, such as *NatureManger* and *TerrainWrapper*. Now when *TerrainWrapper* is ready, we start painting the splat map by painting the base texture for each cell. *TerrainWrapper* has a method for painting triangles, and the Voronoi diagram contains the list of all Voronoi cells that contain cell triangulation; painting each cell with the correct texture based on the cell's biome is then trivial.

The next stage is well known to those familiar with the previous *Road System* described in Section 2.1. We refactored the system to be simpler, faster, better designed, and better formatted, including several minor bug fixes. The first change is that the resulting mesh is produced into the so-called *MeshBuildStructure*, which encapsulates mesh vertices, triangles, and uvs; it easily adds support for merging the meshes and texture atlases, texture arrays, and texture array atlases.

Texture arrays can save runtime performance, but since they cannot be compressed currently by the *Unity Engine*, they increase the build and load time. For that reason, we use them rarely. Another handy feature is so-called composed

side objects. They encapsulate several side objects, for example: composed fans contain fans, a fence, and fence flags. This results in a simpler scene definition when creating the scene manually. Next, we improved both adaptive and regular sampling performance by estimating curve length and sapling it in a smarter way. Another performance optimization is the so-called sapling provider that serves the purpose of caching already computed samplings.

Finally, when we talk about curve sampling, we address two problems: regularly sampled side objects sometimes did not start precisely at the beginning of the curve and did not end at the end of the curve. We address the first problem by starting curve sampling when the side object starts.

The second problem can be addressed only for seamless objects like bushes and guardrails because the root of the problem is that the interval is not divisible by the length of the prefab. Therefore, we handle it by instantiating one additional prefab at the end.

This approach is vital when creating such precise cutouts in the bushes in front of the houses seen in Figure 4.6 or when creating guardrails looping around the whole circuit, such as in Figure 4.8. House entrances also caused the need for improved aligning of the meshed side objects; the previous approach just aligned the last sample with the terrain. The new approach can lerp the alignment based on distance from the end of the given interval; it can partially align several samples on each side if needed to make better immersion.

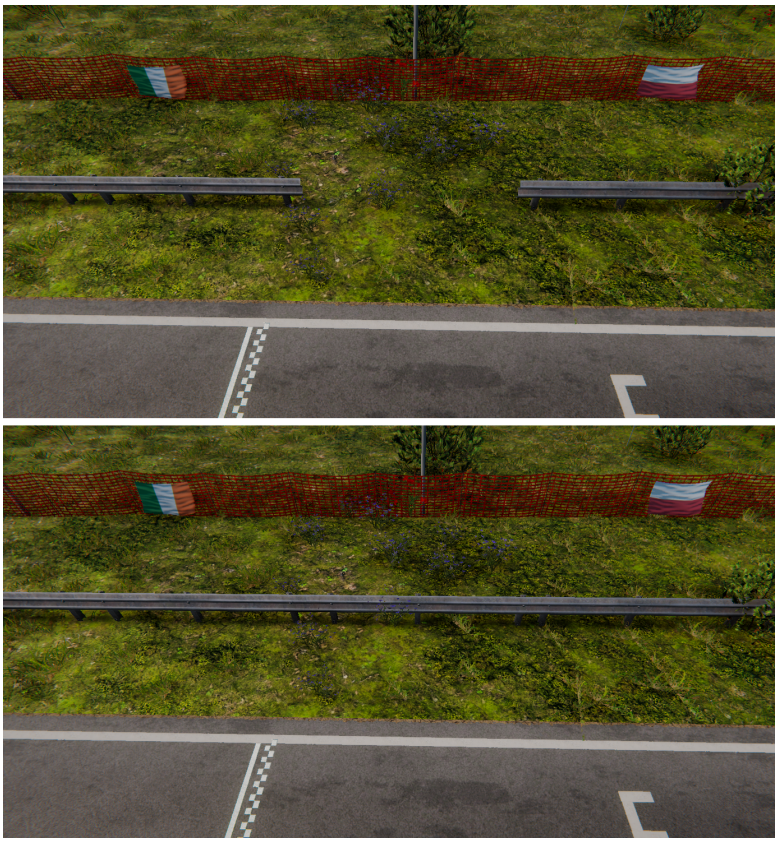

Figure 4.8: Side objects without compensation are at the top, and you can see the side objects do not follow up with each other on the edges of road segments. At the bottom, you can see the same place with side objects end compensation present.

Finally, we added support for spawning trees and individual aligning with terrain for selected prefabs inside composed prefabs, for example, a house with a car and a bush. Source prefabs for those spawned trees are determined by the forest biome at the given place. The example of the composed prefab can be seen in Figure 4.9.

Figure 4.9: Example of composed prefab: including house, bushes, flags, car, and trees.

After that, all *Road System* segments are regenerated and baked in the order in the editor as previously. When drawing generated stuff, we need to split responsibilities between road segments. The nearest road segment draws each feature. We ensure this by finding the nearest *NavigationFrame* class and reading a sibling index of the curve from which it is sampled. First, all references are initialized in the road drawing process; then, the generation is done. We do all generating only in the *RoadEditor* container, ignoring the *LineEditor* container; there are likely no lines drawn if we generate the content.

We start generation by clearing previously generated metadata; we represent this by adding the *generated* bool flag to all *Road System* primitives such as *Line*, *Point*, *Subdivision*, and *Segment*. Roads and paths are next; we already have them generated in the generation structure. We already have roads masked with the main road from the generation to avoid the intersection; if the end was masked, we mark it as automatically connected to the road later. Lastly, we refresh the alignment to the road.

Next, we generate crossroads; they are generated in a standalone step since we can ask the system to generate crossroads alone without generating the rest of the environment. For each end, we find the nearest *GeneratingFrame*, and then we march forwards and back several *GeneratingFrames*. We create two curves between the two frames we have and two ends of the connecting road (left end and right end); to do that, we add subdivisions to the frame samples on the main road curve. We align the curve points tangents with the roads targets, generate road lines as side objects, and add curves to the *GenerationGrid* as side curves,

which results in curbs, sidewalks, bushes, and houses being drawn, but not safety features. This is for realism reasons, not technical. We will expand on this process in the following paragraphs. We add hay bales as a side object to the main road in order not to confuse players with the crossroad; this is demonstrated in Figure 4.12. We can see several generated crossroads by this method in Figure 4.10.

Figure 4.10: Several generated crossroads.

The road segment's main generating loop is next. We generate side objects defined by *GenerationGrid* to each side first. Now is a good time to describe how translating a list of *GeneratingFrames* into side objects works. We go through the list for each side object type one by one for each side separately. For each frame, we have the sample from the curve we are drawing to and a side object presence flag in the frame. If the biome changes, we take it as both the end and the start of the side object

When we have the first and the last *GeneratingFrame* of the side object, we insert divisions into the curve at the frames' sample positions and draw them. Definitions of side objects are provided by *SegmentProvider*, which is the only place where side object codes and material codes are directly in the code. *SegmentProvider*'s parameters are: side object type, biome, and start and end frame. We count biome change as an end and start because side objects can change depending on the biome; we improve the algorithm by checking if side objects are the same and merging them if they are. The changing side object properties are visible in Figure 4.11.

Figure 4.11: Side object properties can change on biome edge.

The next step is adding support for generating certain side objects around the whole circuit. This task introduces us to *DrawSegmentMinusRoad()* function. This function takes the segment we give to it and draws it, excluding all areas where a crossroad is. We do this by finding all crossroad intervals and subtracting them one by one from the argument interval. The hardest part is implementing *LineInterval* subtracting functions; we just need to analyze all possible combinations of two-interval that can (non)overlap - there are only five possible combinations. A simple example of this crucial functionality demonstrated on fans around the circuit is in Figure 4.12.

Figure 4.12: *DrawSegmentMinusRoad()* function - the fans are cut out from the road, and hay bales are added.

In the following step, we iterate through all curves, select roads, and use *GenerationGrid* to generate sidewalks, bushes, and houses in the same ways as in lines creating crossroads. We reuse precisely the same logic for drawing and most of the logic as when we calculated the main road. We calculate *normalized angles*, relative distance from the edge, houses, and house entrances in the same way. We highlight curbs only at the crossroads to make them more exciting and normal sidewalks in the cities. All culling is done in the same way as on the main road. We finish side roads by adding cars between the first two control points using the *DrawSegmentMinusRoad()* function to add a little bit of immersion; fans typically park owned cars not so far from the track.

Next, we draw treelines on all edges, where fields on both sides surround a path. Finally, we want to draw bushes around all city cells to close all house gardens. We calculate the Voronoi inner hull, with distances from edges dependent on edge width. We determine this by calculating each edge separately; we extend the edge in length and then subtract all volumes of other edges extended in length that are extended into infinity on the side not facing the cell middle as well. We do this only in cells that contain at least one edge that contains asphalt road. The generating diagram provides this computation. Finally, we use only edges from the hull that are not asphalt, dissolved, and close enough to the track and mask them with the main road extended by the sidewalk. We can see bushes generated by this algorithm in Figure 4.13.

Figure 4.13: Bushes placed using inner hull algorithm.

We then bake the road segment as previously, extended by additional features discussed throughout this thesis. One visual improvement the *Road System* received is that the asphalt shader now supports shading the ideal line based on the vertex information. One thing to remark is that the road edits the underlying terrain splat map using *TerrainWrapper*, and the masking triangles fill the mask for tree masking. After road baking is done, we create meshes into the mesh container. Support for texture atlases, texture arrays, and texture atlas arrays is done using *MeshBuildStructure* by either remapping uv sets or adding

an additional uv set. Subsequently, terrain editing is finished by painting small tree lines along a few randomly selected edges, improving the immersion and dirt texture under nature paths. Finally, we paint the paths connecting houses with roads with clay; this information is simple to obtain from *GenerationGrid*.

We have finished the terrain editing; we commit the changes. However, unfortunately, it would be pixelated if we committed changes to the splat map as we draw it right now. To combat this issue, we antialiase it with a simple filter. That takes squares of 5×5 pixels and weights them based on the distance to the center. We need to be careful not to antialiase terrain too much; we would lose details then. The square size was not chosen randomly; it is crucial to antialiase the terrain and not destroy its features, as demonstrated in Figure 4.14.

Figure 4.14: Not-antialiased, antialiased, too much antialiased terrain.

Next, we generate checkpoints; we can use a *NavigationGrid* for this task since it is nicely regular, and the checkpoints need to be as wide as the circuit is. Then we draw the global minimap into texture and save it. When we speak about glocal minimap, it is good to mention that thanks to *NavigationGrid*, it is simple to draw cheap local minimap dynamically in UI because we can limit the maximum number of steps thanks to the normalization. The global minimap was previously drawn quite suboptimally since each main curve was marched, and hundreds of circles were drawn on the Bézier curve between consecutive control points. Although it was simple, effective, end surprisingly fast, we want to improve it.

We base the new algorithm on *NavigationGrid*; we iterate through all *NavigationFrames*, and get points on the sides using a fake value of road width computed from the size of the bound of the track. This way, all minimaps have the same style. Then we get points inside the triangulated circuit. We then draw a line in place of the start/finish for obvious gameplay reasons. We can speed it up by skipping a few frames/samples in each iteration. It also proved to be helpful to render minimap in higher resolution and downsample it. The main bottleneck is converting *Texture2Ds* into .png to save size and saving it. This means we can never get under five seconds when generating a minimap unless future research radically changes the approach. We can see such generated minimap in the figure 4.15.

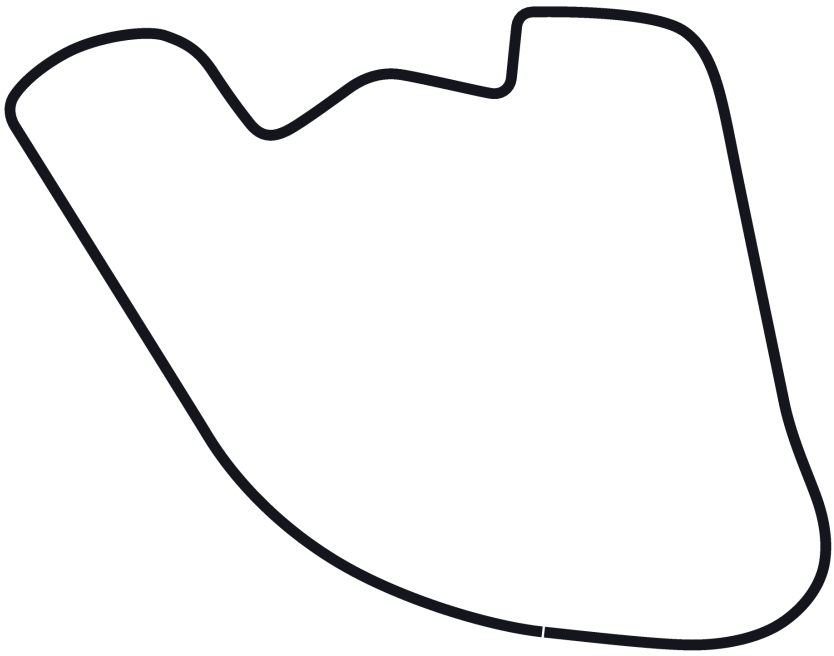

Figure 4.15: Baked, downsampled minimap.

The next part is the naive part of the background generation; this part is only an experiment and proof of concept for future research. The current version is straightforward and is mainly a high-performance validation of the importance of such generated background. We have 3 *HillsTypes*: small, medium, big, and city. The user can select between them or none; in that case, this step is skipped. The city hills type calculates the bound of the circuit + padding and then marches around the bound and spawns skyscrapers randomly, so they follow up with each other. In other cases, we have a prefab of the whole hill/mountain as a mesh with a size 2×2 km, and it is spawned around the track. Although these methods are simple and do not produce logical results in some cases, they proved potential in generating far distant content. The big hills background can be seen in Figure 4.16, and the city background can be seen in 4.17.

Figure 4.16: Big hills background.

Figure 4.17: City background.

4.5 Nature generator

The next component is the nature generator implemented by *NatureManager*. It is the component with the best demonstration of variable quality/density of instances based on gameplay importance, in this case, mainly distance from the track. We need this nature manager to be compatible with manually created tracks because we want to model both real-world tracks and generated maps. For this reason, nature manager is mainly based on the splat map workflow, but we use several features from generating a diagram, such as getting the distance from the track. Furthermore, we want to have support for several nature styles, namely: leafy, mixed, conifers, beech, and pine[2]. In most cases, one map has one biome, but they can have any mix of biomes.

The outline of the algorithm is: generate some excellent points for potential trees; for each point: check if they have desired distance from the track, check if they are placed on the correct splat map, check if they are not inside a mask. We use *Poisson disk sampling* (Särndal et al. [1992]) for point generation; there are other saplings, but we selected this for speed and simplicity. This algorithm also resembles reality very well.

We generate the points inside track bounds with padding intersected with terrain bounds. Padding will be defined shortly. Next, we implement variable density based on distance. Then, we implement so-called cascades, defined by maximum distance and sampling density.

In practice, we take the track bounds, extend them to each side by the cascade distance and check for each point if it is inside the cascade. We implement support for any number of cascades by enforcing the distance to be for a given cascade between the maximum distance of the previous cascade and the current cascade. Finally, we implement the splat map rules by checking at each sample if the value in the splat map at a specific position is high enough. This is provided

[2]Names of the biomes are based on the names given to the assets by third-party creators.

by *TerrainWrapper* API, which handles all complex terrain manipulation to hide terrain implementation details from the rest of the code.

The next problem we tackle is forests being visibly thinner at the more distant places; this is especially visible at the forest's edges. For this reason, we detect forest edges. We need a similar structure as the distance function at the diagram API. We start by filling the grid with zeros when the forest is not present and infinity if it is present. Then propagating values in the same way. What we get is the distance function to the non-forest place. Forest edges are all places with this value below a specific value. At these places, we divide the distance from the track by a constant, which forces more detailed cascades further from the track at the edges while keeping the instance count low, as demonstrated in Figure 4.18.

Figure 4.18: Distance cascades with forest depth. Most of the vegetation is on the edges of the forest and close to the road.

The approach we selected not only saves performance and space it is also based on reality and thus makes the scene feel more natural. We can see such an example in Figure 4.19, where the edge of the forest is denser and with lower trees than the center of the forest.

Figure 4.19: The forest is usually denser at the edges with smaller and wider trees.

Next, we already discussed nature masks a little bit, and their target is to exclude vegetation from certain areas. We again divide the world into cells and have a grid containing bool values marking if any mask is potentially present in the cell, and we have a dictionary, where the cell is the key, and the value is a list of masks. This structure is easy to implement and fast; it is not a bottleneck. The result is demonstrated in Figure 4.20.

Figure 4.20: Mask precision showcase. As we can see, we can have dense vegetation close to the road's edge. We are limited by the float precision and quality of the prefabs. The *Vegetation Studio Pro* mentioned in Chapter 2.2 has a rather poor accuracy.

We implement different forest biomes to give our world a final bit of spice. Each cell has its forest biome that is decided similarly to the biome. We can override all biomes to one selected, which is widespread usage, or mix them randomly based on the seed of the cell. In Figure 4.21 you can see all five biomes mixed.

Figure 4.21: The mix of all five forest biomes.

We support spawning along the line simply by defining such side objects in the *Road System*. The problem is that these trees need to be masked as well. Unfortunately, the mask is not fully initialized yet when the *Road System* is being drawn. Therefore, we define a boolean flag in the side object graph for *GameObject* prefabs named *maskable*. We mark any of these *GameObjects* to be checked later in our generation process. We check them now during the vegetation spawning and delete them if they are to be masked. Point spawning is very rare; for that reason, we edit instances using Unity Editor, which is perfect for such tasks. We need to serialize these drawn trees because *superbake* destroys them. We store each tree's prefab reference, position, and scale. Rotation is randomized. This approach proved to be very effective.

In the super baker phase, we serialize all *GameObjects* into *Matrix4x4* and delete most of them. However, it is very performance wasteful, especially with high instance counts of the trees, because many memory allocations are done. For that reason, we bypass spawning the *GameObjects*, that are physically deleted from the scene - not all of them, but many are. Instead of that, we construct *Matrix4x4* ourselves. Creating a transform matrix is simple and one of the most fundamental algorithms in real-time computer graphics. First, we create a scale matrix, multiply it with a rotation matrix, and finally, a translation matrix. This makes the whole process fast.

Next, we generate the grass. Before optimization, it took 4 minutes on a small track to spawn grass alone, and it took up 400 MB of disk space. The key is to have the grass in the *NoGameObject* workflow and to bypass instancing process. The final step is speeding up Poisson disk sampling since it is the bottleneck now

in grass spawning. We solve this by generating only a single tile much smaller than the track, 100×100 meters proved to be excellent value, and tile the tile repeatedly. As a result, the grass is now generated in under one second on most tracks.

Figure 4.22: Flowers and grass.

We want to have flowers in our grass to make our world more colorful, but without increasing draw calls. We achieve this by creating a texture array, 4×4 in our case, and using world position as a seed using a custom shader. Then, when we spawn the grass, move it by a tiny bit into position, producing the correct array index. This distance is only a few millimeters at most, and we can check the masking afterward. We now only need to decide what spawns where. Usually, flowers are mostly in small patches. Then, we introduce so-called *GrassRules*; they work based on splat maps, distance cascades, and a list of Perlin noises with thresholds. We can think about it as sub-rules; we find the first sub-rule that fulfills the threshold condition at the given point and shade the grass with the associated texture array index. Lastly, flowers are more prominent than grass, and we can do the same as we did with the forest edges but inversed: we force cascades closer to the track if the grass/flower is less prominent, depending on the bool flag in the sub-rule. The possible result can be seen in Figure 4.22.

4.6 Superbaker

The final step of the whole algorithm is superbake, making the whole scene small, fast to load, and adding a tiny bit of performance since some *Collider*s, *Component*s, and *GameObject*s are removed entirely. Superbake is quite a simple process. First, we superbake fans; all fans are located in fans containers; we iterate through all *GameObject*s in all of them, save *Transform Matrix4x4* and delete the *GameObject*.

Then, we initialize *GPU Crowd Instancer* in the *Start()* method with those matrices and randomize animation clips[3] and starting frame for each instance. We similarly do standard *GameObjects*; the main difference is that we unpack those prefabs because they can be nested, and the parent one does not have to be a *GPU Instancer instance*.

Initialization in runtime is the same except for two minor details. The first one is that we can merge several *Matrix4x4* containers into one in a low-quality setting. For example, all tree prefabs into one, or we can skip instancing some objects entirely. Second, we update the color of the ideal line by updating the material if it is enabled to reflect if throttle or break should be pressed based o the AI computation. Ideal line updating is the only part of this system that runs in the runtime. The remaining *Collider*s are visualized in the figure 4.23. The same spot but in the runtime can be seen in Figure 4.24.

[3]Random from suitable clips.

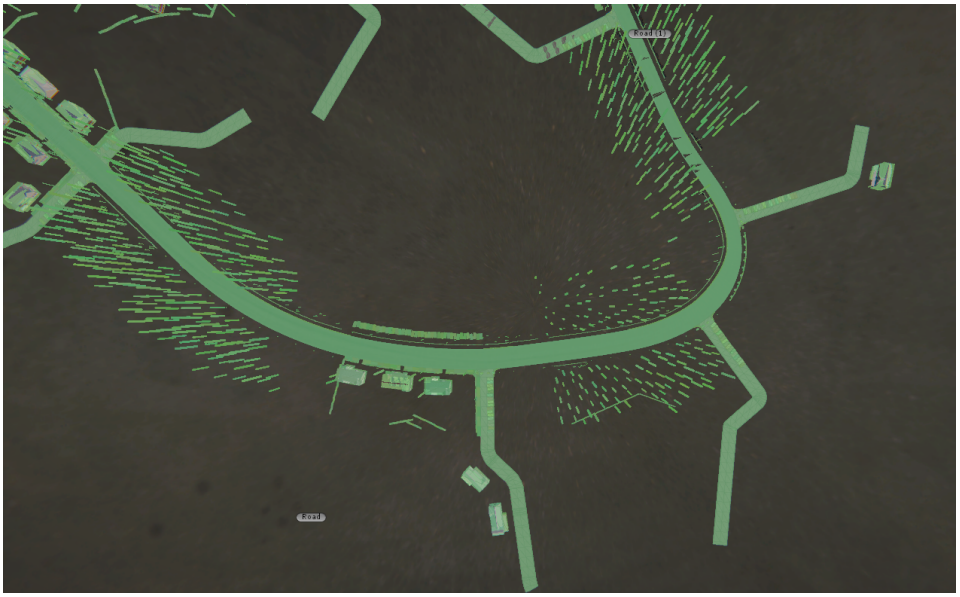

Figure 4.23: *Collider*s after superbake. You can see each *Collider* in the green, and the background is brown. The outline of the road is visible, and around it are *Collider*s, in this case mainly the trees, that are closer than 50 meters to the track.

Figure 4.24: Figure 4.23 rendered from the same spot, but at runtime.

4.7 Results

You can see the produced results in the attachment files of this thesis. There are screenshots and videos of the editing process available. We opted not to include compiled demo as it would be too big because of the size of the source assets and terrain texture arrays. Instead, you can download the demo game, which is not part of the thesis named *Engine Evolution 2021*[4]. The handling of the game and the free camera is described in the attachment B together with several interesting levels for this work.

You can also find the source codes relevant to this thesis in the attachment files. In Attachment A are high-level overviews of the most essential classes.

The attachments of the thesis are include two short videos. Please note that the game's framerate was locked at 60 FPS to improve performance stability. Both videos are only short introductions to the system and only small parts of the system in order to minimize video sizes.

[4] `https://store.steampowered.com/app/1589770/Engine_Evolution_2021/`

5. Evaluation

In this chapter, we go over the feedback of both players and artists on the produced system. We also discuss performance and possible future work.

5.1 User feedback

In the following paragraphs, we would like to present several feedbacks from the users of the demo game. Please note that these players often refer to the state of the demo game before our system as the old system and after as the new system. Please excuse any grammatical mistakes; we did not want to alter the feedback.

It is important to stress that the track surroundings are not the core part of the gameplay of the demo game, so players are unlikely to speak about our system. We selected several players who played the demo game for an extended period and asked them, which means that the players we asked were already a bit biased. Nonetheless, players like the improvements to the demo game a lot.

One stated: *"When I first saw the version of the game with the new system, I say, omg, that looks amazing."* and: *"Now the grass looks really realistic and along with small bushes the immediate vicinity of the track looks perfect."* Finally, he stated: *"The new system has improved performance, and overall, everything works much more smoothly."* The second player stated: *"The new landscape is amazing, more beautiful than the Land of King Miroslav :D."*

Although the sample of players was relatively small, we can conclude that the track surroundings are much more detailed and more performant in the demo game than previously. A more detailed study may be beneficial in directing future research in the right direction.

5.2 Artist feedback

The system was tested on two users of the previous system. Both users found the new system very beneficial and achieved all required goals.

The first user is very biased since he is the author of the thesis. He found the system very simple and intuitive, and the average time to design the track surroundings he was satisfied with was always under five minutes. This time is insignificant to the time spent with the track layout design. He further noted that when he was designing tracks manually, it was always very difficult to come up with ideas on how to make the track original and where to place stuff. It also took a lot of time to create just one village, and the quality was still worse than with the new system. The old process was completely non-interactive, while the new one was completely interactive and very flexible. The system scales well with the size of the maps and enables the artist to create a map if he does not have any idea, some idea, or a good idea of how the track surroundings should look.

The second user stated that he did not have the patience to create the track surroundings in the old workflow. Since he cared mainly about the quality of the racing, he lacked ideas and time to create track surroundings. The new system completely solved this problem, and he was happy with the default settings of the

generator, thus taking him thirty seconds to create his first track surroundings. He then played with the result only for fun, liking all designs. The second artist is a student of the law, so he has a very little technical background. His introduction to the system took about 10 minutes.

The target of simplifying the track surroundings design for the user was, in our opinion, achieved. Thanks to default parameters, it is usually enough to just let the generation process to do its work. Although our experimentes included just two artists, we think it is enough since the system will always be used only by a limited number of previously trained artists.

5.3 Performance

We managed to achieve less than 250 draw calls per frame in Unity 2021.3.0 LTS, HDRP 12.1.6. The 250 draw calls are shown by frame debugger while having two shadow-cascades, volumetric lighting, post-processing, and some other features. The target framerate is at least 120 Hz on Nvidia 2070 (Mobile). Our peak performance is less than 200 draw calls and 170 FPS.

During the development process, we used *Vegetation Studio Pro*. After removing it, our frame increased by 10%. We estimate that nature rendering times decreased by 35% based on measurements on several devices.

There was no significant critical feedback on performance and quality when the reviewer understood that he spoke about a demo for a diploma thesis. However, some players provided negative feedback when they did not read the description and thought they had downloaded some AAA game from a major games studio; similar negative reviews are present even at these AAA games; we interpret this as our work starting to be confused with AAA games at first glance by some.

5.4 Future work

Although the system is already capable of generating the track surroundings and achieves all goals we have set, several topics we came up with during the development can prove to be great topics for future research.

First such topic is terrain editing. This thesis already contained API for aligning terrain under the road, which proved useful on several occasions. Since editing terrain heightmap is a destructive operation in *Unity Engine* workflow, we opted not to present it in the main text of this thesis. This thesis is already capable of aligning terrain under the road. Heightmap editing works in the same way as splat map editing; there is an API that converts the triangle into the terrain space, and then the texture is edited. It will be possible to base the terrain modeling module on this API in the future. The following possible extensions are splat mapping based on the terrain slope and terrain modeling based on road shape. Currently, the road is painted on top of the terrain and follows it.

The next problem is smaller and is rather a software engineering problem. *GameObject*s in the *NoGameObject* workflow are serialized as *Matrix4x4* structures. Some *GameObjects* could be defined only by position and scale, which could be four float values; the downside is that the matrix would be constructed

in *Start()* method, which could take too much time. Further experiments are needed to determine the usability of this approach.

Next, the current *Road System* is based on curves and does not have a proper API to spawn a single prefab at a certain point. The *SideObjectGraph* is already up to the task of defining, but the spawning is complex. It is currently handled it with masked spawning[1], but a proper API would be very beneficial in the future. Future API should also support *NoGameObject* spawning in the same fashion as *NatureManager* and Raycast support for determining y position.

Also, each track already has its own style provided by track design, biomes, forest biomes, and background hills. This could be enhanced by stylizing sidewalks, curbs, safety features, and other side objects. The system is so far able to spawn individual houses. It would be great if it were able to generate city blocks with variable building sizes. This would most likely require generating meshes in the more advanced version of the system, perhaps textures as well. Some streets contain parking slots or grass strips between the road and the sidewalk; this also cannot be modeled by the current generator.

Finally, a professional racing circuit can contain escape zones on the outside of the corners, giving the vehicle time to slow down before impact. However, modeling those is hard since the barrier distance from the track changes; hence big changes in the logic generating safety features will be required.

5.5 Conclusion

All targets declared at the beginning of the thesis were successfully achieved. Both the players and artists were happy with the system; they rated it better in quality and performance and significantly better in terms of usability than the previous system and work-flow. The system is also a good starting point for future research on further styles, terrain generation, adding more details, and even better automation.

[1] Masked spawning is defined by offset and period. Offset causes first samples to be ignored, and period causes samples to be skipped.

Bibliography

Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, sep 1991. ISSN 0360-0300. doi: 10.1145/116873.116880. URL https://doi.org/10.1145/116873.116880.

Paul de Casteljau. de casteljau's algorithm. 1959.

Jonas Freiknecht and Wolfgang Effelsberg. A survey on the procedural generation of virtual worlds. *Multimodal Technologies and Interaction*, 1(4), 2017. ISSN 2414-4088. doi: 10.3390/mti1040027. URL https://www.mdpi.com/2414-4088/1/4/27.

Ulrich Göhner. Procedural race track generation for domain randomization procedural race track generation for domain randomization. 07 2020.

Ian Hudson. Race track generator, 2019. URL https://i-hudson.github.io/projects/2019-02-02-Race-track-Generator/.

Petr Šimůnek. Content creation tools for 3D racing games, 2020.

Daniele Loiacono, 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*, 3:245 – 259, 10 2011. doi: 10.1109/TCIAIG.2011.2163692.

Gustavo Maciel. Procedural racetrack generation, 2020. URL https://bitesofcode.wordpress.com/2020/04/09/procedural-racetrack-generation/.

MasterPixel3D. Race track generator, 2018. URL https://assetstore.unity.com/packages/3d/environments/roadways/race-track-generator-113050.

Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, jul 1985. ISSN 0097-8930. doi: 10.1145/325165.325247. URL https://doi.org/10.1145/325165.325247.

C.E. Särndal, B. Swensson, and J.H. Wretman. Model assisted survey sampling. 1992. URL https://books.google.cz/books?id=MWCzngEACAAJ.

Awesome Technologies. Vegetation studio pro, 2018a. URL https://assetstore.unity.com/packages/tools/terrain/vegetation-studio-pro-131835.

GurBu Technologies. Gpu instancer, 2018b. URL https://assetstore.unity.com/packages/tools/utilities/gpu-instancer-117566.

GurBu Technologies. Gpu instancer - crowd animations, 2019. URL https://assetstore.unity.com/packages/tools/animation/gpu-instancer-crowd-animations-145114.

Attachments

A. High-level overview of the classes in the attachment

This section introduces the main components of our implementation that are available in the attached files in the folder *Implementation*.

RoadGroup, and RoadDataSave

These scripts are based on the *Road System*, there was a change in almost all *Road System* scripts, but these stand out. The *RoadDataSave* was extended by API, enabling drawing curves and dice objects on a high level. One such example can be the function *DrawLine()*, taking two points and a side object definition. Or function *DrawAsphaltRoad()*, taking the type of the road and point array as arguments.

The *RoadGroup* has undergone major modifications, and it is now responsible for orchestrating and storing references to each part of the system.

NavigationGrid

During this thesis, we are building features that help orient the world space. *NavigationGrid* is such a feature; it is an abstraction over all road segments and caches important road information in each sampled point inside the class called *NavigationGridFrame*. *NavigationGrid* is mainly focused on AI support, and it represents the area of the road. Please note, due to *Unity Engine* serialization details; the classes are named *NavigationSupport* and *NavigationSupportFrame* in the attached scripts; we very much apologize for this complication.

GenerationGrid

GenerationGrid is similar to *NavigationGrid*, but each side of the road is sampled independently instead of sampling the road center. Each sample is represented with *GenerationGridFrame* class. Its primary purpose is to help us translate generated data into input data for *RoadDataSave* classes. *GenerationGrid* represents the edges of the road. It is responsible for analyzing the shape of the road edge and generating safety features, curbs, fans, and more.

MyVoronoiGraph

MyVoronoiGraph class generates and provides access to all essential information about the generated world. At any point in the world space, you can get biome at that place, distance from the track, distance from the edge, or get the bounding box of the track with padding and much more. *MyVoronoiGraph* contains definitions for *MyVoronoi* cells, *MyEdge* and *MyVertex*, that are located at the edge intersections. All these features contain many functions. For example, the *MyVoronoi* class can calculate the inside hull of the cell. The *MyEdge* class can return the minimum distance from the track. The *MyVertex* class has such function as finding the closest neighboring *MyVertex* to the track. All these structures

have more functions, including determining the parenting *Road System* segment and much more.

TerrainWrapper

The *TerrainWrapper* is the component that handles all the operations with the terrain. Its core functionality is to transform the points between the world space and terrain texture local space. It contains features such as getting the texture at a given point and painting terrain inside the given triangle. Since working with the terrain is a destructive operation, some features are only experimental, such as aligning the terrain with a certain triangle.

NatureManager

NatureManager is a *MonoBehavour* class responsible for efficiently generating trees and grass. It mainly focuses on the variable density of the vegetation based on its distance from the track. It also realizes manually drawn instances of vegetation and handles vegetation masking.

InstancingSerializator(s)

InstancingSerializator and *InstancingSerializatorGO* are *MonoBehavour* classes responsible for efficient serialization, *Collider*s culling, and scene disk-space optimization. *InstancingSerializator* class focuses on animated fans. *InstancingSerializatorGO* focuses on other *GameObject*s. They do nearly the whole rendering in the game's runtime.

SideObjectGraph

There needs to be a drag&drop interface, where we define side objects, tree rules, et cetera. *SideObjectGraph* class is the place where this is possible. It also handles the *Side Object Graph* manipulation.

B. Playable demo

First, you need to download the demo game from the *Steam* named *Engine Evolution 2021*[1] after that, you launch the game, hit the "Auto-login via Steam" button, and fill in your profile information. Next, you go to the main menu and hit the "Race" button, and select the "Time Trial" game mode. Finally, you need to select a track and hit the "Start!" button.

You can interact with the demo in several ways. The first one is driving the vehicle around the track with the arrows. The up arrow is the throttle, down arrow is a break. The Left and right arrows are for turning. You can see settings by hitting Esc, going into settings, then input and editing input settings there. Most controllers and driving wheels are supported out of the box as well. While in settings, in the Game tab, you can enable Advanced spectator mode (DO NOT USE)[2], which enables a free spectating camera. The spectating camera is then turned on by hitting the key P and then H. You can then rotate the camera when holding the right mouse button and moving with the mouse. You can move around the map freely, including down under terrain and inside the objects with W, A, S, and D. You can move up and down using Q and E. Finally, you can adjust the field of view by either pressing Keypad + and Keypad -.

The demo contains five scenes:

- Hořice — the real track from the City of Hořice. It was the original track for the demo game. It was mainly drawn manually without using the generation tools. It has the most advanced nature, which was generated while using all five forest biomes. Please keep in mind that most logical errors are done mainly by the user and are the state of the art that the user can do. The whole map took thousands of man-hours during the nearly six years of development; all houses were drawn manually; hence it is not fair to compare this aspect with other maps.
- Village — this track was done using the *Road System*; it took dozens of hours to create. The nature was generated, and all other features were drawn manually.
- Hills — the Hills track received the best rating from the generated maps. It uses all generating features of our system, and most features were developed and tested on this map. In the end, a great level of detail was achieved on this map; several features were demonstrated only in parts of the Hořice map or never at all. Such as lowered sidewalks in front of houses, bushes around the houses, and dirt paths.
- Outskirts — the whole track is generated in a similar way to the Hills map.
- High plateau — this map is the main validation of the process of generating the surroundings of the track. The generation itself uses just the default parameters, and they were never altered; hence we can claim that the track generation took no time.

[1] `https://store.steampowered.com/app/1589770/Engine_Evolution_2021/`

[2] This warning is included in order to discourage the players of the demo game from turning on this feature since it can be confusing to inexperienced players.

- Ghost — is the final map of the demo; this map was the map generated by the second artist. The second artist is a student of the law, so he has a very little technical background.

In the attached files, you can find two short videos: *DemoBuild.mp4* captures one lap around Hills track. In addition, the *Editor.mp4* captures some of the core functionalities of the system in the *Unity Editor*.