Student report, not an official Aalborg University document

Designer-Friendly Methods for Manual Changes of Procedurally Modeled Terrains

Semester Project by Toke Wivelsted

from 01-02-2016 to 31-05-2016

AALBORG UNIVERSITET

Fifth study year
Computer Science and Software
Selma Lagerlöfsvej 300

Title:

Designer-Friendly Methods for Manual Changes of Procedurally Modeled Terrains

Project period:

Spring semester 2016

Project group:

dpt105f16

Participants:

Toke A. Wivelsted

Supervisors:

Martin Kraus

Hard Copies: 0

Total Pages: 68

Appendix: 15

Completed: 31-05-2016

Abstract:

The following report was written as part of a 10th semester project at Aalborg University, Computer Science.
Procedural Modeling (PM) is an algorithmic technique for generation of virtual content. When used for generating terrains, the outcome of PM can be hard to predict, often causing designers to choose a purely manual approach instead. However, the time saved by using PM can be quite significant, and it is therefore worth considering how the procedures can be steered and made more predictable.

Based on the framework prototype created during my 9th semester project, this report proposes a solution which lets the designer steer the modeling process. This is implemented by having the designer specify locations on the terrain, where different ecosystems should appear. A Voronoi diagram is utilized for storing the placement of ecosystems. The ecosystems, which are also created by the designer, hold elevation values and information about terrain features. The characteristics of the various ecosystems are then extracted from the Voronoi diagram when the terrain is procedurally generated. This method is preferable to the sketch-based approach implemented during 9th semester, as the influenced areas of the placed ecosystems are immediately visible. The designer is no longer moved from the terrain construction process, and the trial and error testing of mapping ecosystems to colors are no longer present. Furthermore, the generated Voronoi diagram can be utilized by known interpolation methods to determine the final elevation values of the terrain.

Two *tools* are introduced, intended to help the designer alter a procedurally generated terrain: a *brush* for sculpting the terrain, and, *a river tool* for carving rivers into the terrain. Terrain alterations caused by these tools are stored in *layers*, which can be toggled on or off, or have their strength adjusted.

The method for steering the terrain generation process, in conjunction with the custom tools and layer functionality, makes the procedural approach a viable alternative to manual terrain creation.

Foreword

This report was made at Aalborg University as part of the 10th semester of Computer Science by Toke A. Wivelsted.

References for sources used in the report are shown as numbers surrounded by square brackets, e.g. [7]. The report itself is composed in LaTeX and the source code for the framework is written in C# as an extension to the Unity editor. A list of abbreviations used in the report is given in Appendix A. Likewise, a guide on how to install and use the framework is supplied in Appendix D.

I would like to thank my project supervisor *Martin Kraus* for his valuable feedback and help throughout the semester.

Toke A. Wivelsted _____

Contents

1 Introduction

This report covers the continuation of my 9th semester project [11], *A Framework for Sketch-Based Procedural Modeling of Terrains for Use in Computer Games*. Whenever the *base prototype* is mentioned, it is a reference to the framework and its features created during 9th semester, whereas *prototype* is the product being developed during the 10th semester project. The prototype is simply the next iteration of the framework, using the base prototype and all its features as a foundation.

Computer games and the virtual worlds they present are increasingly becoming larger and more complex. One of the major components, if not the biggest, of a virtual worlds is the terrain. A terrain can have many functions in a computer game, primarily facilitating a foundation for the player to move across. Manually constructing a terrains can be a monumental task, requiring artistic expertise and countless hours of meticulous refinement.

PM is a technique for automatic generation of virtual content, based on a few adjustable parameters. PM can be used for constructing entire terrains intended for computer games, almost instantaneously, which was demonstrated with the base prototype. However, leaving the creation process entirely to a system of procedures is not an optimal solution, as the generated terrain often needs a human touch to follow design requirements.

The base prototype proved that a sketch-based approach utilizing PM is not only possible, but also highly desirable as it can significantly reduce the time spent on creating terrains for computer games, while letting the designer influence the outcome by supplying a 2D sketch.

In this semester report, I look at how to expand upon further involving the user in the terrain creation process. Existing tools and graphical concepts known from other applications are examined and custom built prototype extensions are implemented in an effort to empower the designer. The three major components added are: enhanced ecosystem placement using Voronoi diagrams (Section 3.3), layers for storing manual changes to the terrain (Section 3.4), and the custom tools for performing the changes on the terrain (Section 3.5 and 3.6).

The prototype focuses on the simplest terrain representation and scope [11, Section 2.2-2.3], being a confined height field, as was the case during the 9th semester.

The novelty of this project lies with how Voronoi diagrams utilize ecosystem specifications [11, Section 3.3.2] to construct a terrain procedurally. Furthermore, with the introduction of layers, changes performed by the designer are stored separately from the procedural terrain, and can be reapplied if needed.

1.1 The Problem

The major problems with using a purely procedural approach for constructing terrains intended for computer games are: limited artistic control and loss of manual changes. These two problems will now be introduced and explained.

1.1.1 Retaining Artistic Control

Retaining artistic control refers to the ability to translate an idea into a virtual model. For example, a designer may want to model a wast desert where an oasis containing a single tree is located in the middle of the terrain. Taking the manual approach, everything in the terrain would have to be sculpted to mimic a desert, a cumbersome task, but the vision could be perfectly recreated. With a purely procedural approach, a desert could be created within seconds, but the full vision would be very hard to realize accurately, as the outcome of even a single procedure is incredibly difficult to predict. The goal is to find an optimal combination of the two extremes, benefiting from the power of procedural modeling while retaining artistic control.

The base prototype alleviated this problem a little, by allowing the designer to sketch where the characteristics of specific ecosystems [11, Section 3.3.2] should appear.

1.1.2 Preserving Manual Changes

When generating a terrain using PM, there is almost always a need to refine or correct some areas in order for the terrain to follow design requirements. For example, a designer may want to place a castle on the terrain and have it surrounded by a moat. The designer finds a suitable location for the castle and starts carving out the moat in the terrain. However, if the terrain is ever regenerated from the underlaying procedures, the manual changes previously performed on the terrain are lost. Regeneration of the terrain is necessary when changes are made to the procedures. There can by many reasons to update the procedures: changes are made to better reflect nature, new assets are added, perhaps a different resolution is wanted.

Preserving the manual changes is of utmost importance, but how these changes are stored, and reapplied to a terrain is an issue that must be resolved.

1.1.3 Summary

The lack of control over PM and the generated outcome is often the reason why a manual approach is taken instead. PM is simply too unpredictable, and even minor changes to the procedures can have huge impact on the final outcome. However, as was proven with the base prototype, using PM to create the terrain can significantly reduce the amount of time needed. Furthermore, the outcome can be directed with the help of a ecosystem-sketch and by defining elaborate rules for the procedures.

Although a sketch-based approach have proven to be effective, it does not remedy the aforementioned issues; it is a step toward furthering the use of PM and retaining artistic control, but the current process is somewhat cumbersome. Consequently, the design and generation process in the prototype are revisited in an effort to streamline the process and improve user control.

1.2 Hypothesis

With the problems introduced the hypothesis can be stated:

Procedural modeling, of virtual terrains intended for computer games, becomes a viable alternative to manual creation when designer-friendly methods, for performing changes on the procedurally generated terrain, are introduced.

To help verify or disprove the hypothesis, some questions will need to be answered:

- How can artistic control be retained?
  - What are some of the existing tools for manipulating virtual terrains, and how do they help improve artistic control?
  - How can the design process implemented in the base prototype be improved?
- How can manual changes be preserved and survive the regeneration process?

While answering the questions, I try to find solutions that promote user control and intuitiveness. The prototype should be powerful enough to create realistic terrains for computer games, yet simple enough for any one to use. It should be straightforward to construct and maintain a terrain, without restricting the designer.

2 Analysis

The purpose of the analysis chapter is to research existing ideas and solutions to the previously introduced problems. Papers trying to combat the artistic limitations induced by procedural modeling are examined. Likewise, numerous existing tools and frameworks for manipulating terrains are examined. Lastly, the framework developed during 9th semester and its features are reexamined with the new problems in mind.

2.1 Brushes

The most readily available tool in graphics applications is the brush. The brush is also part of the default terrain editor in Unity [10], where different properties of the brush can be set, such as size and strength. The typical brush simply increases or decreases the height values of the area it is applied to. In Unity, it is also possible to select the shape of the brush from a collection of regular and irregular shapes and patterns, shown in Figure 2.1.

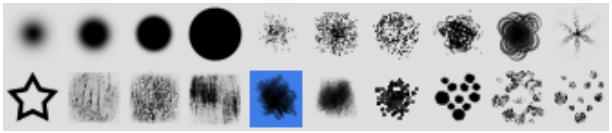


Figure 2.1: The brush shapes available in Unity 5.3.1

Giliam J.P. and Rafael Bidarra introduces procedural brushes [3], which aims to take advantage of the power of procedural modeling, while leaving the overall control to the designer. The brush samples a noise function (noise functions explained in [11, Section 2.4]) to add realistic features to the terrain. The designer can, by varying the size of the brush, choose to have fine control over the sculpting process or let the procedural system apply its output over a greater area. The procedural brush can be seen in Figure 2.2.

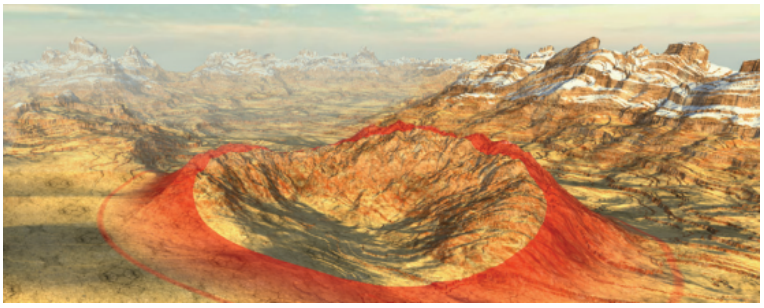


Figure 2.2: The procedural brush being applied to a terrain.

Evaluation

Brushes are the most plentiful tool for terrain manipulation, and with good reason; if the changes caused by the tool are reflected immediately, it feels very natural to use, like drawing a painting. And, like an artist with multiple brushes of varying thickness and bristles, having different shapes for the brush to alter the terrain will benefit the designer.

Using procedural brushes to supply pseudo random values, instead of applying an identical value over an area, is an excellent idea which can also help prevent sharp edges of hastily drawn brush strokes.

2.2 Silhouette Curves

James Gain et al. [4] illustrates a method for sketching terrain deformation. In their implementation, a designer can sketch a silhouette curve over the terrain, which will cause deformation in the landscape to match the curve. An example of using this sketching tool can be seen in Figure 2.3.

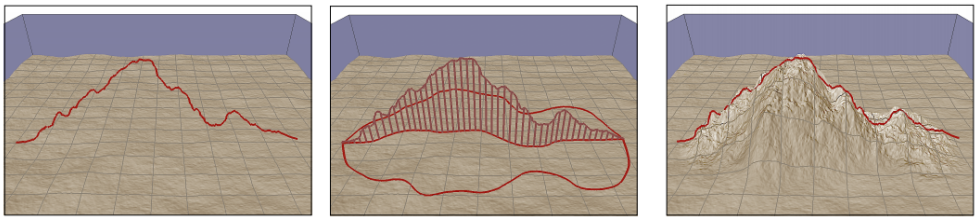


Figure 2.3: The silhouette curve, its shadow, and the resulting deformed terrain.

It is also possible to adjust the area of influence under the curve, i.e. the shadow (width) of the mountain and have several curves influencing one area. Finally, they make it possible to draw custom areas of influence, wherein a curve can be drawn to specify the roughness of the enclosed terrain, shown in Figure 2.4.

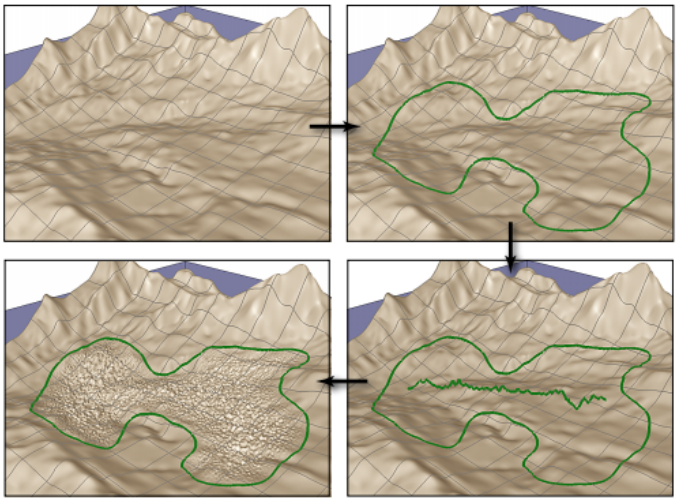


Figure 2.4: Defining roughness for a contained area.

Evaluation

This tool looks like it would be a great asset to a PM framework, though hard to control in a 3D terrain editing environment. In their implementation, only straight curves are possible, meaning a bend in a mountain would have to be accomplished using two or more curves. But without a doubt, this visual tool replicates what is drawn by the designer and would help overcome the hindrances with using PM. As a side note, this tool cannot stand alone and must be able to manipulate an existing terrain generated from noise functions. Otherwise, it would force manual sculpting of the entire terrain, which is what we are trying to avoid.

2.3 Feature Curves

Another method of using curves as a tool is presented by Houssam Hnaidi et al. [5]. A flat surface (2D) is populated with parameterized feature curves, which are sets of bézier splines. Every bézier spline consist of two or more constraint points (minimum two points at extremities), which specifies custom constraints for that point.

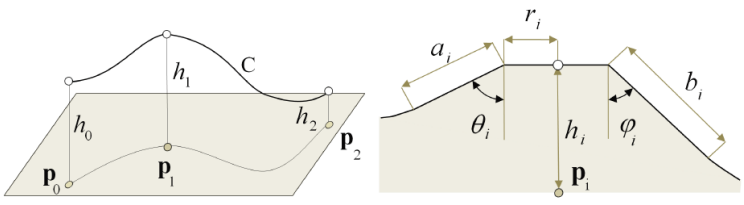


Figure 2.5: Geometric constraints on points of a feature curve.

There are three constraint categories, one for elevation constraints, another for angle constraints, and a third for noise constraints. Elevation constraints are height and radius of influence. Angle constraints are the length and angle of the slope. Noise constraints are used to customize the noise generator, e.g. setting the seed and octave count for a Perlin noise generator.

The designer can choose to constrain a point with none, any, or all of the constraint categories. An example of the geometric constraints can be seen in Figure 2.5, where C is a feature curve, p_i is a constraint point, h_i is an elevation constraint, r_i is the radius of influence, a_i and b_i are the length of the slopes, θ_i and φ_i are the angles of the slopes.

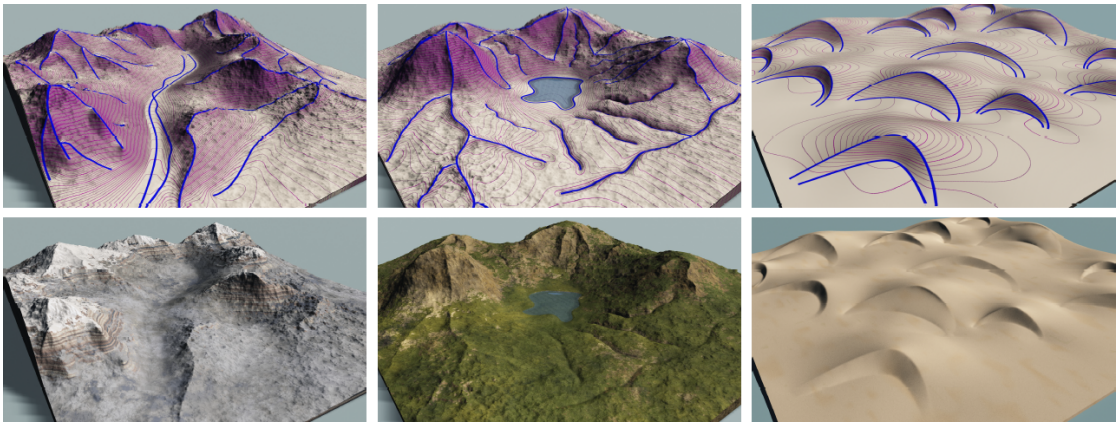


Figure 2.6: Examples of parametrized curves used to generate various terrain features.

Evaluation

This solution uses bézier splines which allow for complex terrain definitions. Figure 2.6 illustrates that interesting and unique terrain features can be modeled with this tool. All of the illustrated examples was generated in less than a second, utilizing a Graphics Processing Unit (GPU). It is clear that this tool enables fine control over terrain details by using constraints, but this also makes it increasingly complex. A designer will have to construct the entire terrain, from a flat surface, by adding feature curves. Every constraint point added, increases the number of possible attributes, which the designer must maintain.

2.4 SketchaWorld

SketchaWorld [9] is a complete framework for generating and modeling terrains (confined height fields). The framework comes with its own rendering engine and a handful of basic tools for selecting and manipulating the terrain and its features. The terrain itself is displayed in a top-down view, taking up the majority of the user interface, which is shown in Figure 2.7. What makes this framework stand out, however, is its implementation of layers. In SketchaWorld, there are five different layer categories: Landscape, Water, Vegetation, Road, and Urban.

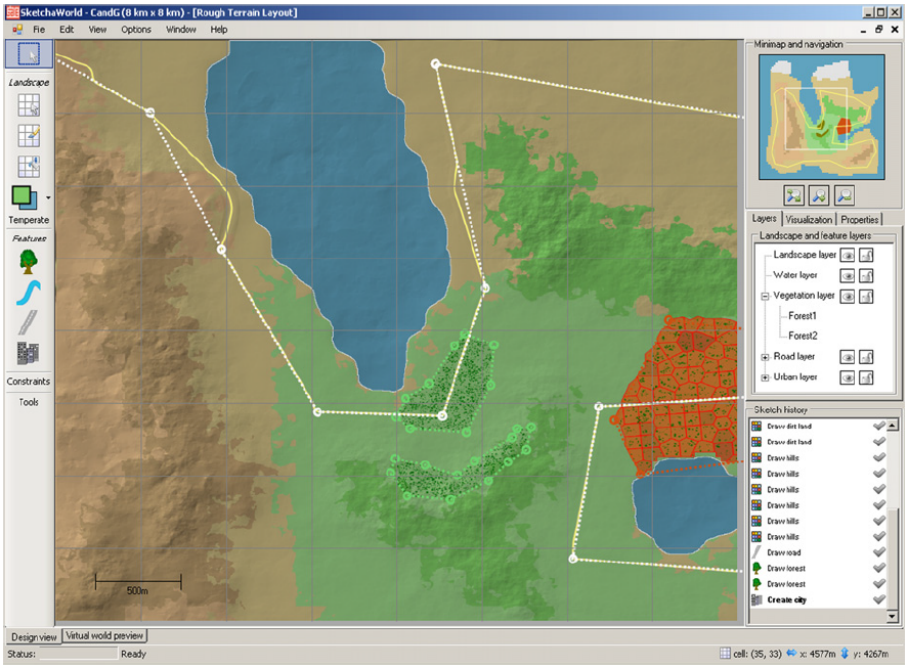


Figure 2.7: The user interface of SketchaWorld, showing the tools on the left side, a minimap, the layers and action history on the right, and with the terrain in the center.

The Landscape layer is a unique layer, and consist of a grid of equally sized squares, each specifying the ecotope of that particular region. Ecotopes holds information about elevation ranges, roughness of terrain and soil details. A brush is used to paint ecotopes on the landscape grid, one ecotope per grid square.

The Water layer is simply a plane extending the entire terrain at a specified elevation, but, it is also capable of holding a subset of other water layers for rivers drawn on the terrain. A river is added by drawing a polyline on the terrain, which is then saved to a new layer.

The Vegetation layer, like the Water layer, is capable of holding a subset of layers, each containing information about a particular piece or group of vegetation, e.g. a forest. Vegetation is added by drawing a region with a polyline starting and ending in the same position, where anything enclosed is populated with a user-specified type of trees. This information is then saved to a new layer, as a child of the Vegetation layer.

The Road and Urban layers are for handling man-made structures, which function similarly to the Vegetation layer, but will be ignored here as this report only focuses on the natural aspects of a terrain. An illustration of how ecotopes, vegetation and water is drawn on the terrain is shown in Figure 2.8.

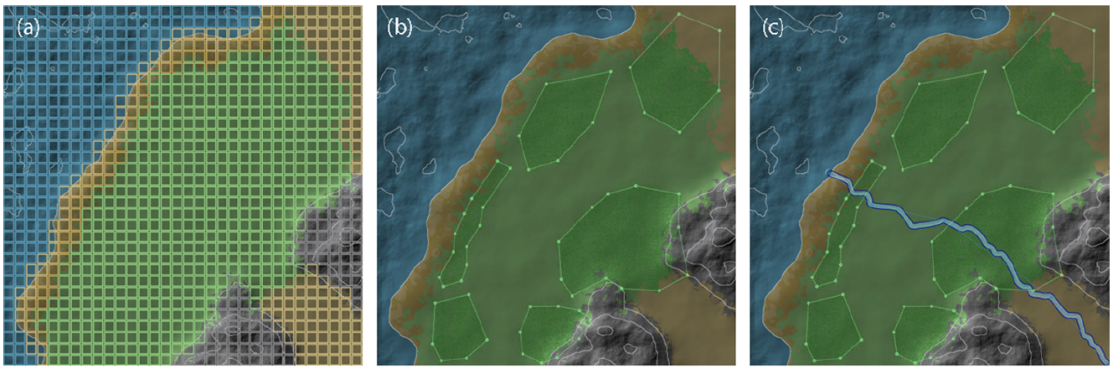


Figure 2.8: Painting ecotopes, vegetation and rivers in SketchaWorld.

The visibility of the five categories can be toggled on and off, and the changes are reflected in the terrain immediately. It is also possible to zoom in and out on the terrain, making it easier to fiddle with details. A minimap gives the designer excellent overview of the entire terrain, and it can be used to quickly center the camera over a selected region.

Evaluation

A lot of effort have been put into making vegetation seem natural, by enabling advanced details of plant species to be adjustable. For example, the density and the preferred soil type can be specified, and noise will automatically be utilized in the placement of trees in order to break up patterns. Likewise, when the rivers are generated based on information stored in the layers, several factors will influence the final path of the river, e.g. high elevation values may cause the river to deviate from its original path.

By dividing the terrain into five categories, it becomes a lot easier to maintain the various features when more and more detail is added. And, by having the different features in individual layers, every manual change to the terrain is effectively stored and can be reapplied if the underlaying landscape is changed and need to be regenerated.

The framework contains a lot of interesting ideas for defining feature areas (vegetation and rivers) and tools which can help the designer (layers, ecotope and vegetation manipulation). But, the framework is also lacking on a few points. It is not possible to toggle visibility of individual layers, only entire categories. The entire terrain must be manually created; noise functions are utilized for ecotopes, but the placement of ecotopes must still be undertaken. By locking the designer to a top-down view, it can become difficult to visualize the scale of terrain features.

The framework is a proposed solution to the problem stated in, *A Proposal for a Procedural Terrain Modelling Framework* [8], a paper by the same authors as of SketchaWorld. At the time of writing, it was not possible to find a digital copy of SketchaWorld, and the website for the product have been discontinued.

2.5 The Framework Prototype

In this section, the framework developed during 9th semester and its features is examined with the purpose of locating components where artistic control can be added or significantly enhanced.

2.5.1 Terrain Manipulation

There was no tools added to the framework for manipulating the terrain after it had been generated. Although Unitys built-in tools could be applied on individual terrain segments, the

overall terrain would contain large seams when tools were used on segment edges. Likewise, any rules specified for ecosystems [11, Section 3.3.2], would be ignored by the default manipulation tools.

The basis of the terrain is created with noise functions and PM, but, without tools for manually manipulating the terrain, visually realizing an idea can be extremely difficult. Therefore, the framework should contain a set of tools for manipulating the terrain, without causing seams in the terrain. These tools are designed and implemented in Section 3.5 and 3.6.

2.5.2 The Ecosystem Sketch and Color Mappings

In the base prototype, the process of constructing a terrain is divided into several steps: ecosystems are created, ecosystems are mapped to specific colors, a sketch is supplied, and finally the terrain is generated. This approach is a bit clunky as the sketch must be created in another application, and, the mapping of colors to ecosystems [11, Section 3.3.3 and 3.4.3] can be difficult and lead to unexpected results, if for example colors of same light intensity are used for two different ecosystems. Every mapped ecosystem would, to some degree, effect the outcome of every position of the terrain, which was not really the intention; there should be no evidence of a frozen tundra ecosystem in the middle of a rainforest ecosystem. Another problem were the transitions between ecosystems. Sharp and sudden color changes in the sketch from one texel to the next would translate poorly to the terrain. A solution to this problem could be to blur the supplied sketch before processing it, but, this would ultimately result in a different sketch than the one designed, possibly resulting in an unwanted terrain. Furthermore, any changes made to the sketch would not be reflected in the terrain until it was regenerated from the sketch, which caused a significant bottleneck in the design process. Instead, having a continuous approach where any change could be seen immediately would definitely improve the artistic control over the outcome.

2.5.3 Summary

The framework has multiple components which would benefit from being upgraded to allow for real-time reflection of alterations. Likewise, some parts of the framework can be greatly improved by enabling the user to visually control the outcome. An alternative to the sketch-based approach and ecosystem mappings is introduced in Section 2.6.

2.6 Voronoi Diagrams with Natural Neighbor Interpolation

This section introduces Voronoi diagrams with natural neighbor interpolation (NNI), which can be used to construct a height field from a point cloud. It is introduced as an alternative to the sketch-based approach taken in the base prototype, where each point in the height field was calculated by combining values stored in a texture with weighted values generated by noise functions [11, Section 3.4.3]. The following description of truncated Voronoi diagrams with NNI is based on a 2010 paper by A. Beutel et al. [2].

A point cloud is a collection of non-uniformly spread points and a single cloud can contain an arbitrary number of points. To create a Voronoi diagram, every point in the cloud is turned into Voronoi cells. A Voronoi cell for point p, contain all points whose euclidean distance is shorter to p than to any other point q in the point cloud:

Let $S = \{p_1, ..., p_n\}$ be a set of n points in $\mathbb{R}^2$. For each point $p \in S$, its Voronoi cell $Vor_S(p)$ is defined as

$$Vor_S(p) = \{x \in \mathbb{R}^2 | \, ||x - p|| \leqslant ||x - q|| \, \forall q \in S\}$$

Interpolation between the points stored in the cloud can be used to extract a height field, i.e. a value for every point in a grid, using an elevation function $h : S \to \mathbb{R}$.

Natural Neighbor Interpolation

When interpolating the height value for a given point x, any number of existing points $p \in S$ could be used to influence the final outcome. With NNI, a Voronoi cell for point x is temporarily constructed, and only the cells in $Vor(S)$ being overlapped by $Vor(x)$ are used to interpolate the result:

$$h(x) = \sum_{p \in S} w_p(x) h(p),$$

where $w_p(x)$ is the area of $Vor(x)$ overlapping $Vor_S(p)$.

Figure 3.6 illustrated a Voronoi diagram and NNI, where a red dot is a point $p \in S$. The blue lines define the Voronoi cell of the individual points, and the shaded cell for point q is the temporary cell added, demonstrating the aforementioned overlapping.

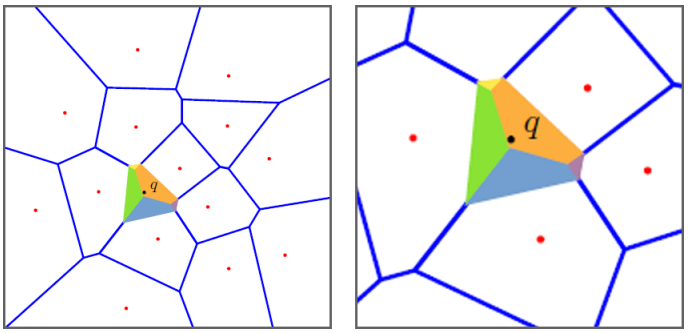


Figure 2.9: A Voronoi diagram using NNI to calculate the value of point q.

Truncated Voronoi Diagrams

A truncated Voronoi diagram consist of Voronoi cells with limited reach. The reach is defined by the radius r:

$$TVor_S(p) = \{x \in \mathbb{R}^2 \,|\, ||x - p|| < r \wedge ||x - p|| \leqslant ||x - q|| \,\forall q \in S\}$$

Having a limited radius of influence can be helpful in certain scenarios, but if all Voronoi cells are truncated, some points may have no natural neighbors.

Evaluation

The intension of analysing truncated Voronoi diagrams with NNI was to find a possible replacement to defining ecosystem placement through sketching. A point $p \in S$ could represent the presence of an ecosystem, and NNI could be used to extract both the influence and noise value of natural neighbors ecosystems for any sampled point. This idea is explained further and tested under the design chapter, Section 3.3.1.

3 Design and Prototyping

In this chapter, features to help better the artistic workflow are introduced. Some of the features are changes while other are additions to the existing framework. The new features are designed with ease-of-use in mind, and are based on the existing resources explored in the analysis chapter. Before implementing the features, their individual purposes and goals are explained in the following *feature overview* section.

3.1 Feature Overview

The most important tasks to implement in this framework iteration are: improvements and tools to help the designer easily form and sculpt the terrain, and, a method for securing manual changes performed and making sure they survive a regeneration process of the terrain. The features being introduced involve solely manipulation of elevation data. No tools or method for painting the terrain or placement of foliage and other objects are introduced in this framework iteration.

3.1.1 Revisiting Ecosystem Extrapolation

One of the best qualities of the base prototype is its ability to hold unique ecosystem specifications. The ecosystems were mapped to user-specified colors, which was then used to determine which ecosystems should influence any given point of the terrain, by sampling a supplied sketch. To remove the problems with ecosystem sketches and color mappings, as explained in 2.5.2, and still take advantage of having custom ecosystems, a different approach is introduced. Individual ecosystems are now placed directly in the Unity scene wherever the designer wants them. For example, the designer may want the northern part of a terrain to contain mountains and the south to contain a desert, so he places the two different ecosystems accordingly. How the individual points of the terrain determines which ecosystem influences it, and by how much, is described in Section 3.3.

3.1.2 Tools

The tools introduced in the following are meant to enable the designer to easily and quickly perform manual elevation changes to an existing terrain. The procedural system is still the main contributor to the terrain generation process, but the tools are just as important, as they give the designer control over the terrain.

While many of the examined tools are clever in their own way, they are also somewhat complicated and difficult to use and maintain. For example, the procedural brush [3] will sample an underlying noise function for elevation values, which can be used to create a natural looking environment. However, it is difficult to use the same tool for creating exact details, as the returned value will vary over distance. The feature curves [5] can be used to generate a complete terrain from scratch, but, the amount of configurable parameters for a single curve makes this a highly specialized task, and every curve added increases the complexity further.

With simplicity in mind, two tools for manipulating elevation values of the terrain are added to the framework. The first tool, the sculpting tool which is implemented in Section 3.5, is a basic brush where properties such as radius and strength are adjustable. Using this tool, both great and small changes can be performed.

The second tool, the river tool, is intended for carving rivers in the terrain. A river is added by selecting a point from where the river will spring and then selecting a point for where it will end. Between these two points are an adjustable bézier spline, and manipulating it will change the path of the river. From an endpoint of a river, any number of new rivers can be added, resulting in a forked river. The river tool is explained further and implementation in Section 3.6.

To enrich the framework, any action performed will not influence the terrain directly, rather, the actions are stored in custom layers. A single layer can be toggled on and off, immediately showing the change of the stored information. Likewise, the transparency of a layer can be altered, causing the effect of the actions to be lessened or increased; a fully transparent layer is the same as an inactive layer and the stored actions will not be applied to the resulting terrain. Layers are implemented in Section 3.4.

3.1.3 Preservation of Manual Changes

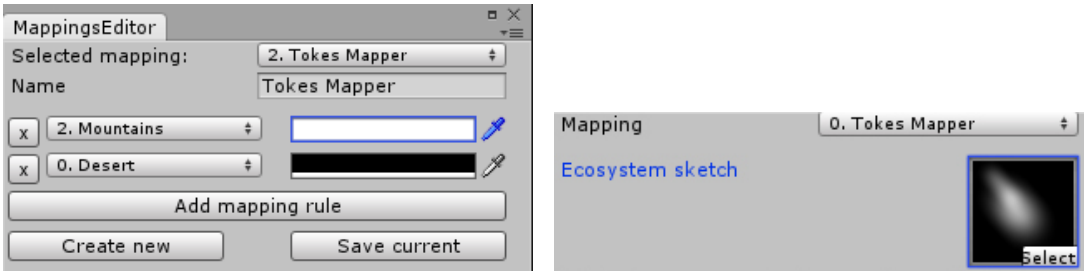
The other important aspect of the framework improvements is to preserve any changes made, i.e. making sure they survive a regeneration of the terrain. By storing the actions performed by tools in layers, the framework have an effective way of distinguishing between data provided by the PM system and data provided by tool actions. How the actions are stored and eventually reapplied to a terrain is covered in Section 3.4.4.

3.2 The Procedural System

Previously, a terrain was generated by following several steps:

1. Create noise schemes to simulate natural terrain features.
2. Create ecosystem specifications, utilizing the noise schemes.
3. Create color mappings for all desired ecosystems.
4. Visit the settings editor and specify details about the terrain, supply a sketch, and finally have the terrain generated.

In an effort to simplify the overall approach, the last two steps, being the mappings editor (Figure 3.1a) and the sketch-supplying (Figure 3.1b), are disabled. By removing these two steps, a significant amount of time can be saved by avoiding color guessing and sketch adjustments. Color guessing in the mappings editor and sketch adjustments in a 3rd party application adds a level of unnecessary abstraction, which does not only require valuable time, but also moves the designer from actual terrain construction to trial and error testing. However, if these steps are removed, a new method for adding a terrain to the Unity scene must be implemented. In Unity, there is a menu for adding 3D objects to the scene, such as spheres, boxes, cylinders and so forth. It would make sense to place the custom terrain in this menu as well, as shown in Figure 3.2.



(a) Mapping colors to ecosystems. (b) Selecting mapping and supplying a sketch.

Figure 3.1: In the base prototype, mapping of colors to ecosystems were done manually, and a colored sketch were supplied to utilize the mappings.

When a user of Unity wants to add a custom terrain to the scene, he selects it from the menu, and an object is automatically placed in the scene in the form of a flat terrain. The details of the terrain can then be changed by selecting the terrain in the hierarchy window. With the base terrain accessible, we can move ahead and start adding the new and improved functionality in the following sections.

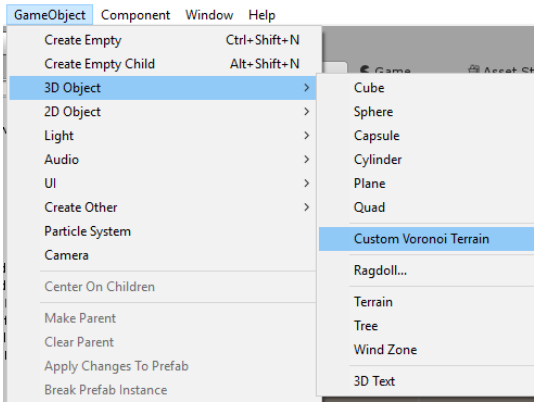


Figure 3.2: The new Voronoi terrain is added to the Unity scene by selecting it in the "GameObject" menu under "3D Objects".

3.3 The Voronoi Terrain

The idea of the new ecosystem handling is to treat them as points in a Voronoi diagram. A user places any number of points in the scene, each referencing a specific ecosystem. The collection of placed ecosystems effectively forms a Voronoi diagram. Any ecosystem can be moved freely around the scene, causing the Voronoi diagram to update accordingly. To help the designer gain an overview of the placed ecosystems, a label for each ecosystem is placed at its position, as illustrated in Figure 3.3.

With the diagram, it is possible to interpolate the elevation values of every position of the terrain. There are many ways to interpolate the elevation values given a Voronoi diagram, and in the following two methods are demonstrated.

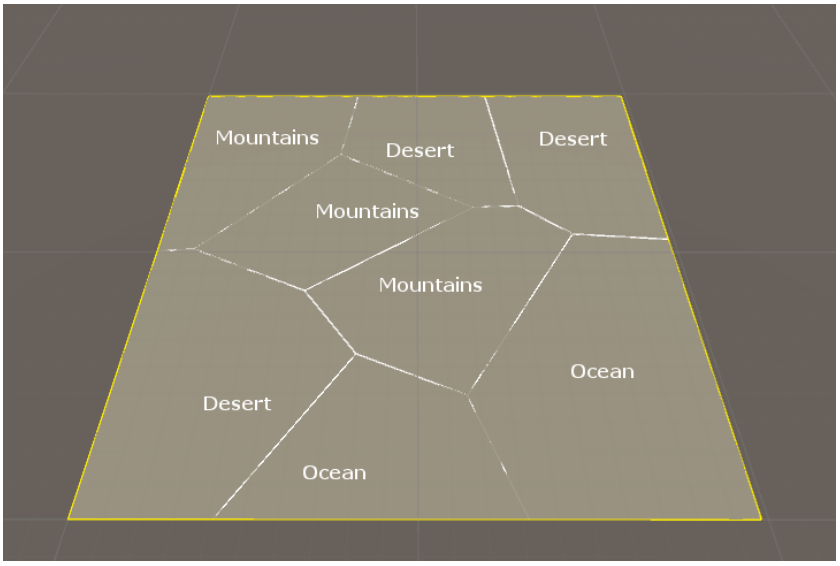


Figure 3.3: A number of ecosystems are placed by the designer. The scene automatically updates and displays a Voronoi diagram overlay.

3.3.1 Natural Neighbor Interpolation

NNI, as introduced in Section 2.6, considers multiple neighbors, depending on the point being processed and the underlaying Voronoi diagram, resulting in high quality ecosystem transitions. NNI does not look at a specific amount of neighbors, instead, NNI looks at the ones who are naturally intruding on the point being processed.

To use NNI for generating a terrain, we will again start out with the basic Voronoi diagram, as previously shown in Figure 3.3. The purpose of the Voronoi diagram is to let every point of the terrain query it to know the position of the ecosystem they belong to. To calculate the elevation values of the terrain, a point q must make a temporary copy of the basic Voronoi diagram and add itself to it. The resulting temporary diagram is illustrated, for a single point of the terrain, in Figure 3.4.

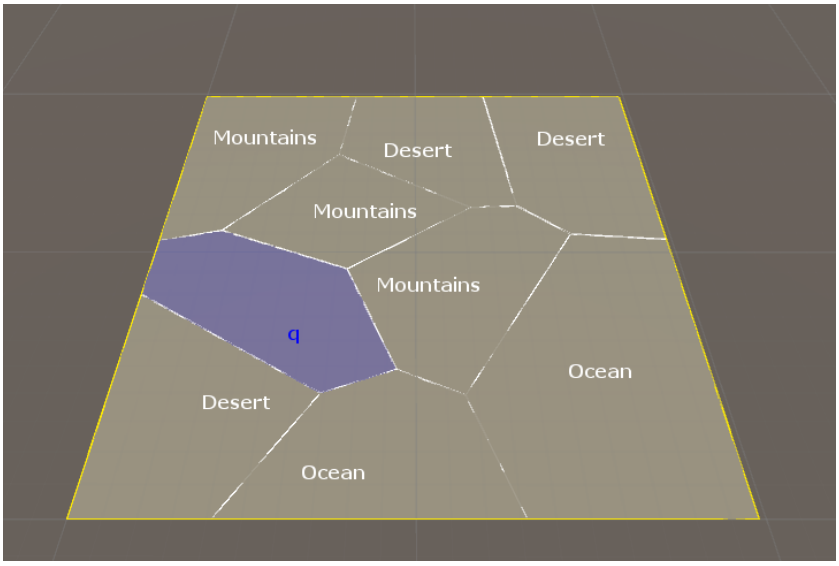


Figure 3.4: The temporary Voronoi diagram; the result of adding a point q, to the basic Voronoi diagram.

Visually, by overlapping the two diagrams, it becomes clear which neighboring ecosystems should affect point q, and by how much. The two diagrams can be seen in Figure 3.5, where the overlapped area of a neighboring cell corresponds to the influence of the associated ecosystem. In the demonstrated example, it would mean that the desert ecosystem should be the main contributor covering roughly 60%, whereas the mountain ecosystem would be about 35%, and lastly the ocean ecosystem of about 5%.

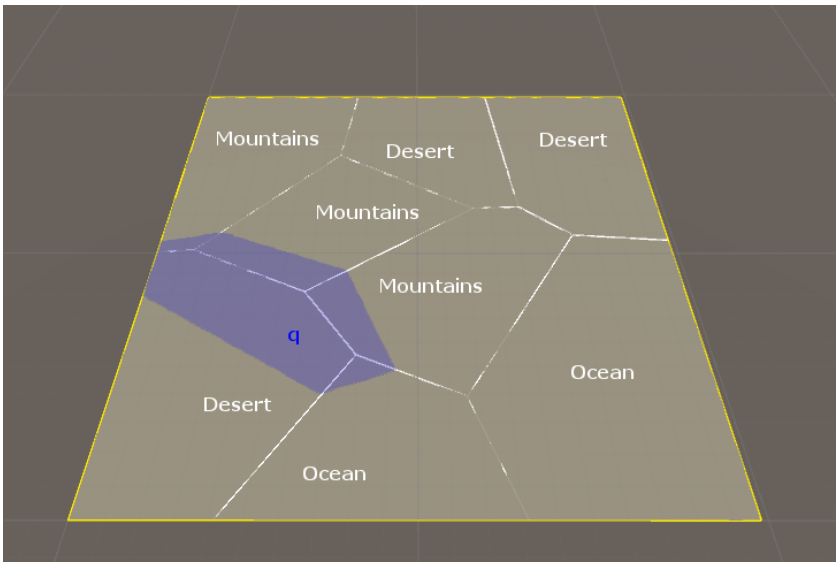


Figure 3.5: The basic Voronoi diagram being overlapped by the temporary Voronoi diagram.

The final elevation value of point q is determined by querying the different ecosystems for elevation values and multiplying their results with the influence value for that particular ecosystem:

$$h(q) = Desert(q) * .6 + Mountains(q) * .35 + Ocean(q) * .05$$

Programmatically, we need to first determine which points of the terrain falls within the cell of point q. Then, with this collection of points, we can query the basic Voronoi diagram to determine how many points each ecosystem governs.

The result of using NNI for every point of the terrain can be seen in Figure 3.6.

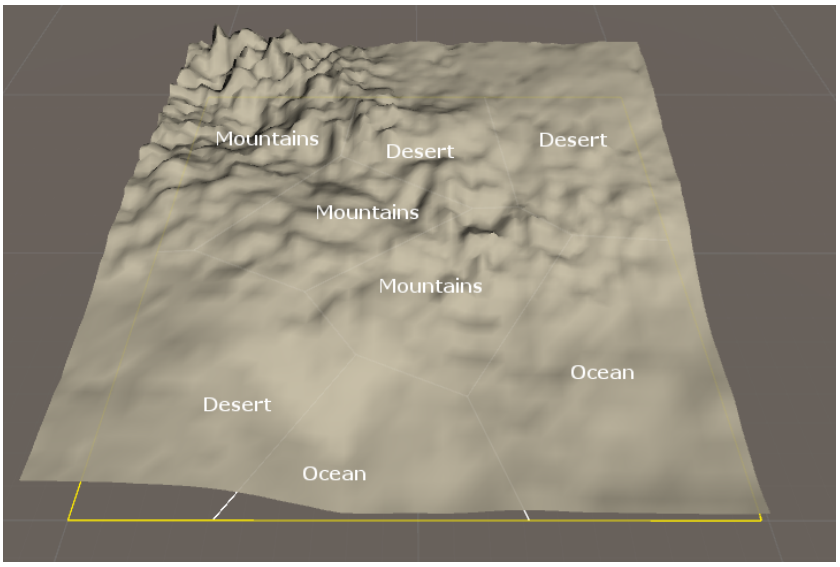


Figure 3.6: A terrain generated using Natural Neighbor Interpolation.

3.3.2 Inverse Distance Weighting

Inverse Distance Weighting (IDW) is another method for interpolating the terrain elevation values. A weighted average of distances is used to determine degree of ecosystem influence. IDW, as introduced by Donald Shepard [7], can be used to construct a terrain from irregularly-spaced data points. These data points corresponds to the ecosystems placed by the user. A placed ecosystem will influence every point of the terrain by some degree, and the closer an ecosystem is to a terrain point, the greater its influence will be. To calculate the degree of influence of placed ecosystems, a slightly adjusted IDW is used.

For a pure IDW, let

- $D \in \mathbb{R}^3$ be a finite number N of triplets (x_i, y_i, z_i), where x_i and y_i are the coordinates and z_i is the elevation value of data point D_i.
- $P \in \mathbb{R}^2$ be the reference point, i.e. the point we need to determine the elevation value of.
- $d[P, D_i]$, shortened to d_i, is the Cartesian distance between reference point P and data point D_i.
- $u \in R \mid 0 < u$ be named the power parameter. The greater u is, the greater the influence of nearby data points are.

Then, to determine the elevation value of a reference point P:

$$f_1(P) = \begin{cases} \dfrac{\sum_{i=1}^{n} (d_i)^{-u} z_i}{\sum_{i=1}^{n} (d_i)^{-u}}, & \text{if } d_i \neq 0 \textit{ for all } D_i (u > 0) \\ z_i, & \text{if } d_i = 0 \textit{ for some } D_i \end{cases} \tag{3.1}$$

In order to make this work for user-placed ecosystems, we only need to specify how the noise function associated with a particular ecosystem is queried. The problem is that the elevation value z_i, will vary depending on both the data point D_i and the reference point P, not only D_i. To solve this, let

- $E(P)$ be the elevation value returned from querying the noise function associated with ecosystem E given reference point P.
- E_i be the ecosystem referenced by data point D_i

Then, we simply replace z_i with $E_i(P)$, as shown in Equation 3.2 to determine the interpolated value of P. For the implementation in Unity, the power parameter u is adjustable, and in Figure 3.7 a terrain is generated with the power parameter set to 3.

$$f_2(P) = \begin{cases} \dfrac{\sum_{i=1}^{n} (d_i)^{-u} E_i(P)}{\sum_{i=1}^{n} (d_i)^{-u}}, & \text{if } d_i \neq 0 \textit{ for all } D_i (u > 0) \\ E_i(P), & \text{if } d_i = 0 \textit{ for some } D_i \end{cases} \tag{3.2}$$

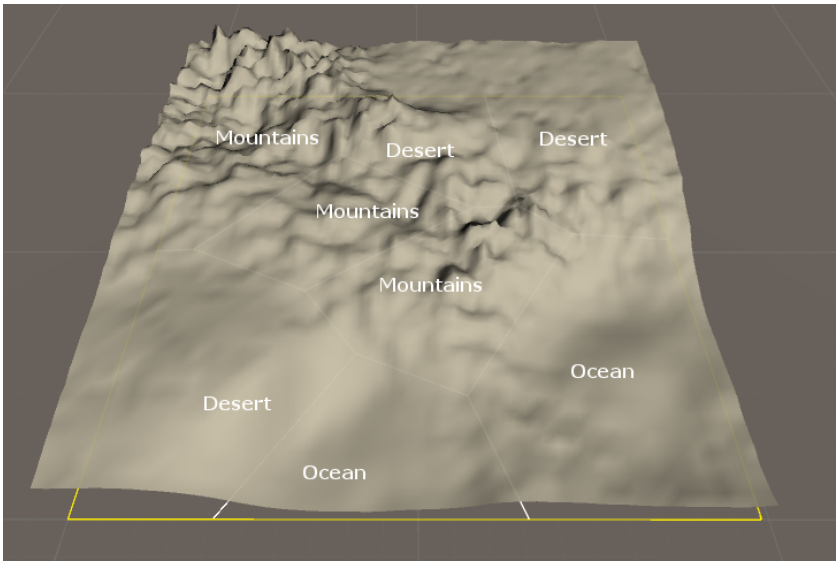


Figure 3.7: A terrain generated using Inverse Distance Weighting with a power parameter value of 3.

The terrain generated using IDW, shown in Figure 3.7, is very similar to the terrain generated using NNI, shown in Figure 3.6. The IDW is, however, computationally much faster than using NNI.

Further Improvements

In his paper [7], Shepard identifies some weaknesses with using pure IDW, the first being the computational requirements for handling large data sets, and secondly, directions of known data points is not taken into account.

To lower the computational requirements, the number of processed data points must be lowered. As only nearby data points are significant when using IDW, computational requirements can be lowered by ignoring distant data points. Therefore, only the n nearest data points within radius r of reference point P should be considered.

The second weakness is that the direction of data points are ignored. Shepard wants to include the direction of data points to determine their influence. Figure 3.8 illustrates that even though the distances between P and the involved data points are identical in the two examples, their interpolated average should not be. A data point in the "shadow" of another should be significantly less influential than the one causing the shadow. In the example, D_1 should be more influential on the left-hand side than on the right-hand side. The opposite should be true for D_3. D_2 would hold the same influence for both sides in the example given.

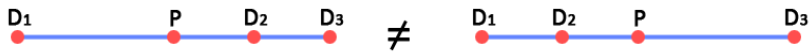


Figure 3.8: A data point in the "shadow" of another should be significantly less influential than the one causing the shadow.

These improvements have not been implemented in the prototype due to time constraints, but it would be interesting to see the differences between the pure and the improved version of IDW, in a future framework iteration.

3.3.3 Summary

In addition to the two introduced methods, Nearest Neighbor Interpolation (NeNI) - a method restricted to only considering the nearest neighbor - was also tested as an alternate interpolation option for terrain generation. Serious effort was put in to try and make the NeNI method applicable for terrain generation as it is computationally much faster than its counterparts. However, the limit of only looking at the nearest neighbor meant that the result ultimately became undesirable, and no apparent application could be found for it. The discarded interpolation method can be viewed in Appendix B, where it is being demonstrated on the same Voronoi diagram as the two other interpolation methods.

A smooth transition between ecosystems is wanted, which is what both NNI and IDW delivers. NNI gives the best result but takes a relatively long time to process (which may not be an issue), whereas IDW is fast and can be tweaked with the power parameter to specify the intensity of how ecosystems blend together. For both methods, it is possible to define the interpolation curve or chose a predefined shape, e.g. linear or s-curve.

3.4 Layers

The primary purpose of implementing layers is to have a means of storing manual actions and reapplying them to a regenerated terrain. Having layers brings many benefits and opens up for

framework expansions. For example, with layers it would be possible to implement manipulation of performed actions, such as duplicating, rotating or scaling everything stored in a layer with a single command. Having layers also provide a safety net for the designer; a subset of actions performed on the terrain can be easily toggled on and off, which would not be possible if changes were performed directly on the terrain. Another expansion could be to store and display the history of performed actions, making it possible to easily backtrack or toggle visibility of individual actions. For this framework iteration, a layer only store the elevation information and nothing about the which tool were used or how many actions the individual values are the result of. Likewise, other information could be stored in a layer, such as foliage and rubble placement or soil characteristics, but the scope of the iteration only allow for handling of elevation values.

3.4.1 Layer Types

Typically, in image processing applications, such as Adobe Photoshop [1], the content contained in a layer will effectively exclude any overlapped content in subsequent layers from being visible. Lowering the opacity of such a layer will allow overlapped content, in subsequent layers, to be partially visible. This classic approach to handling layers is referred to as *synergistic*. Synergistic layers will utilize the opacity of individual layers to realize an outcome. Changing the opacity of a layer scales the stored elevation data. The layer order determine the outcome, and changing the order can result in entirely different terrains.

In a 3D environment it can be problematic to visually highlight the content of a single layer, making it difficult to determine if content on another layer will overlap and take precedence. For this reason I introduce the *additive* layer type. The content stored in an additive layer will always be visible, independent of layer ordering. The opacity of these layers function as a multiplier attribute, determining the strength of the values stored within. Changing the opacity scales the elevation values stored in the layer.

With the additive approach, it becomes easy to duplicate and relocate layers without worrying about whether other layers will overlap. For example, a game designer may want to place recognizable identical landmarks at certain locations, which would be easy to apply using copies of additive layers. Anything made from additive layers is also possible to make with synergistic layers, the purpose of the former is to be less complex and easier to manage. The framework will implement both types of layers for designer convenience.

3.4.2 Storing Elevation Values in a Layer

Any action (change in elevation) performed by a tool needs to be stored on the active layer. A basic 2D texture is used to store these elevation values, and every texel on the texture corresponds to a point on the terrain. Every texel of the texture have four components, or "channels": Red, Green, Blue, and Alpha (RGBA). The value of all four channels is limited to any real number in the range [0;1]. The different color channels are utilized to store various information, as introduced in the following sections. If the designer uses a tool to construct a mountain, the elevation data of the mountain is stored in the texels of the active layer corresponding to the points of the terrain where it was sculpted.

3.4.3 Order of Evaluation

This section describes the process of how multiple layers are combined with a procedurally generated terrain. The final terrain should be determined by iterating through the stack of layers, top-down, combined with the procedurally generated base terrain. Two different approaches are now introduced: static and dynamic. These methods vary only on *synergistic* layers, as the *additive* layers are managed identically: sample additive layers and add their values to the final elevation product.

Static Evaluation

This method strives to minimize the amount of computation needed by keeping track of the opacity of all terrain point while iterating through the layers. Once a point is fully opaque, the corresponding texel is skipped in all following layers. Only when all layers have been processed, will the procedural system be queried, and only if the point is not already fully opaque.

Using this approach requires layers to store the full elevation value of user processed points. Meaning, if a tool is used to carve a volcano out of a mountain, the stored value is the original mountain elevation plus the value subtracted by the tool.

This method have one serious drawback: if the procedural formula is changed and the terrain regenerated, there are visual inconsistencies where the terrain elevation have changed compared to what was originally used for the layers. However, the procedural system is queried far less often, which could be a benefit if the procedural specification are not changed after initialization.

Dynamic Evaluation

With dynamic evaluation, the goal is to make the information stored in the layers re-usable after changes are performed to the procedural system. Any action performed by a sculpting tool have a *change in elevation* associated with it, and it is this value that is stored for affected terrain points. As a change can be either positive or negative, we need to utilize one of the texel channels to hold this information. We use the green channel to store the value 1 for additions and value 0 for subtractions. Using this approach, it would still be possible to keep track of opacities and skip texels accordingly. However, if a regeneration of the terrain is required, all terrain points will have to query the procedural system for establishing base values, which can be quite computationally expensive.

Summary

The two mentioned methods boil down to how elevation values are stored. Either the combined value of procedural queries and performed actions are stored, or, the performed actions alone are stored.

To decide on which evaluation method to use, we first consider how layers and ecosystems specifications are utilized in the framework.

- Creating a terrain is an artistic task, and as such, we cannot expect the designer to have every ecosystem specification planned out and created before the modeling process begins.
- Creating a terrain is an iterative process; a general guideline and style specification may exist, but the details are implemented and changed to fit other game elements which may not be set in stone.
- A procedurally generated terrain often needs a considerably amount of manual tweaking to fit game criteria, and as such, we must expect layers to be heavily utilized.
- Manual changes performed on a terrain may not fit a regenerated terrain if the ecosystem specifications are changed.

Although actions stored in a layer may not fit a changed ecosystem specification, there's a greater chance of being able to rearrange the content of these layers if actions are the only thing stored. With these considerations and assumptions, it would be best to use the dynamic approach.

3.4.4 Implementation

In this section I introduce the base functionality of layers and show how they are implemented in the prototype.

Functionality Overview

There are a few basic features which should be implemented to support the wanted functionality. These features are:

- It must be possible to add a new layers to a terrain.
- It must be possible to toggle between layers, i.e. allowing the user to select which layer is active.
- It must be possible to alter the layer order.
- It must be possible to toggle the visibility of layers, i.e. active or inactive.
- Furthermore, it must be possible to change the opacity of a layer.

User Interface

The user interface (UI) for the list of layers is implemented as a component to the terrain game-element in Unity. Meaning, the layers will only be visible when the terrain is the active game-element. This allows for multiple terrains to exist in the same scene, while hiding the layer information when it is not needed. The implemented layer list and the options for individual layers can be seen in Figure 3.9.

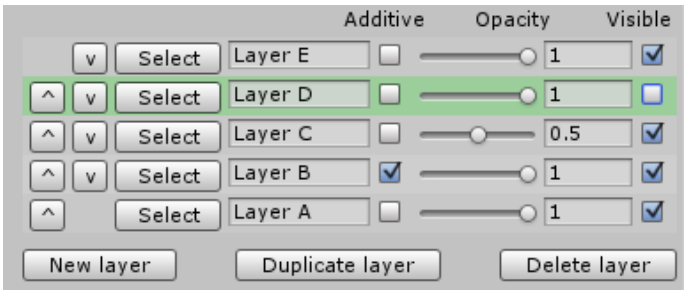


Figure 3.9: A list of five layers created for a terrain, showing the different options for individual layers. The ordering of layers is changed by using the left-most buttons. Setting the active layer is done by clicking the *select* button, and, the active layer will have a green background. A layer can be labeled for convenience.

Reading Layer Data

We must now consider how the elevation values stored in the set of layers are read and combined. Only visible layers are processed, and, a texel on a layer is only queried if the accumulated opacity for that texel on preceding layers does not sum to one, i.e. the texel is not fully opaque. The elevation value of individual texels will vary depending on layer content and layer opacity.

Let

- L be set of n ordered *visible* layers, where $n > 0$.
- O_i be the custom opacity specified for the i'th layer of L, where $0 < i \leqslant n$ and $0 \leqslant O_i \leqslant 1$.
- $O_i(x, y)$ be the opacity of the texel located at position (x,y) in the i'th layer, see Equation 3.3.

- $E_i(x, y)$ be the elevation value stored in the i'th layer at texel (x,y).
- $g_i(x, y)$ be a direct reference to the green channel of texel (x,y) of layer L_i, holding 0 indicating a subtraction and 1 indicating addition.
- $G_i(x, y)$ be the multiplier value indicating whether the stored elevation value of texel (x,y) is an addition or subtraction, see Equation 3.4.

$$O_i(x, y) = \begin{cases} 1, & \text{if } E_i(x, y) > 0 \\ 0, & \text{if } E_i(x, y) = 0 \end{cases} \tag{3.3}$$

$$G_i(x, y) = \begin{cases} 1, & \text{if } g_i(x, y) = 1 \\ -1, & \text{if } g_i(x, y) < 1 \end{cases} \tag{3.4}$$

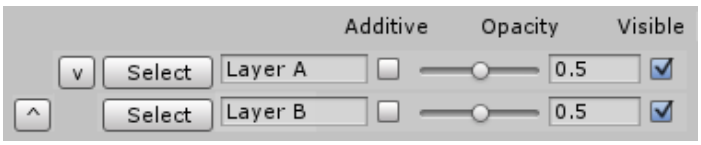
Then, the elevation value for a texel (x,y) is determined by accumulating the opacity of the layers. The accumulation is processed from layer L_1 to layer L_n, where layer L_1 is the front-most layer occluding subsequent layers. The final elevation value of terrain position (x,y), as stored in the layers, is $f_3(x, y, 1, 0)$:

$$f_3(x, y, i, o) = \begin{cases} G_i(x, y)\ E_i(x, y)\ \alpha + f_3(x, y, i + 1, o + \alpha), & \text{if } o < 1 \\ 0, & \text{if } o = 1,\ or\ i > n \end{cases} \tag{3.5}$$

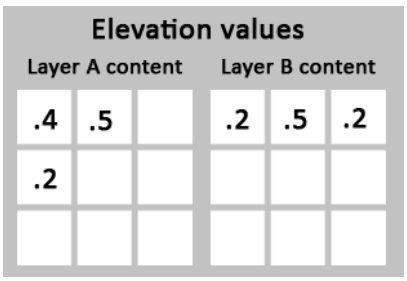
where o is the accumulated opacity and α is the weight being determined for texel (x,y) on the i'th layer:

$$\alpha = (1 - o)O_iO_i(x, y)$$

An illustrated example is now provided to help explain this process further. We start out with two layers as shown below:



Each layer consist of 3x3 texels in this example, and the elevation values (all of which are additions) stored in these are:



A layer can contain empty texels, i.e. points on the terrain that a particular layer does not influence. We then start iterating through the layers, while keeping track of the accumulated opacity:

Accumulated opacity								
Before layer A			After layer A			After layer B		
0	0	0	.5	.5	0	.75	.75	.5
0	0	0	.5	0	0	.5	0	0
0	0	0	0	0	0	0	0	0

As can be seen, only texels which contain elevation values are included in the opacity accumulation. The weight for individual layers are based on their own opacity and the accumulated opacity of preceding layers. The resulting elevation values after both layers have been processed are then:

Resulting elevation		
.25	.375	.1
.1	0	0
0	0	0

That is, for position $(0, 0)$ of the terrain, we calculate its elevation value, as stored in the layers, as:

$$\begin{gathered} f_3(0,0,1,0) \\ = 1 * .4 * ((1-0) * .5 * 1) + f_2(0,0,2,.5) \\ = .2 + 1 * .2((1-.5) * .5 * 1) \\ = \underline{.25} \end{gathered}$$

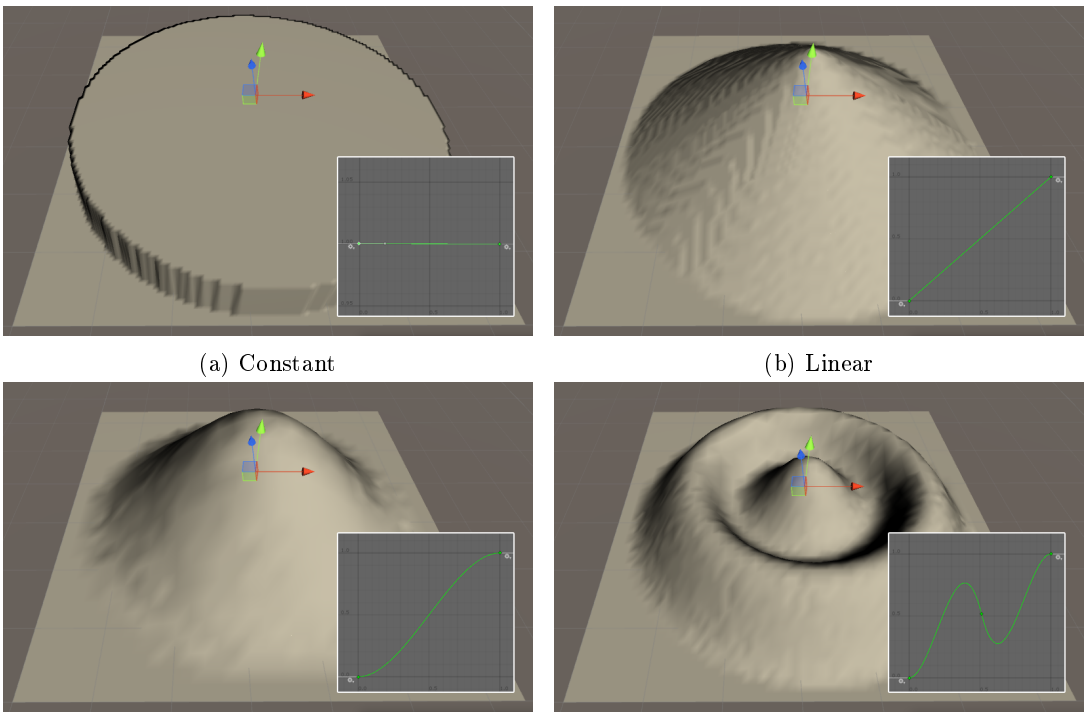
3.5 The Sculpting Tool

The sculpting tool (ST) is the primary means to make quick changes to the procedurally generated terrain. The purpose of the ST is to provide the user with an instrument to make both large- and small-scale changes. Although brush patterns, as introduced in Section 2.1, would be a great asset to the ST, only the simplest brush shape is implemented, as a proof of concept. The simplest brush pattern is a circle, where the encompassed area is manipulated.

3.5.1 Functionality

The tool is implemented much like the default sculpting tool in Unity, i.e. when moving the mouse cursor a semi-transparent disc will display the influence area, and clicking anywhere on the terrain will raise or lower the influences area by some amount. The amount raised/lowered is determined by a *strength* parameter, and, the radius of the disc is determined by a *radius* parameter. However, altering all of the influenced area by an equal amount is typically not desirable, as it would result in steep edges at the circumference. To solve this, an interpolation

curve is added to the ST. Most often, a s-curve is wanted for interpolation, resulting in a smooth mound when raising the terrain. But, to further expand on user control and customization, the curve is manipulative. The curve defines the brush's opacity, as a function of the radius to the center of the brush. Figure 3.10 demonstrates various curves and their effect on a flat terrain.



(a) Constant (b) Linear

(c) S-curve (d) User-defined curve

Figure 3.10: Various interpolation curves for the sculpting tool, and the result of applying a single click on a flat terrain.

3.5.2 Elevation Operations

The ST must be able to both add and subtract elevation values. With these action we simply need to change the elevation value stored in the layer, by adding or subtracting a value for every affected texels. As the elevation value stored in a texel is not signed, we utilize the *Green* channel to store whether the elevation value must be added or subtracted. A *Green* value of 0 means the action must be subtracted, and a value of 1 means it must be added. When updating the value of a texel, we must first compare the performed action type with the stored action type and alter the value accordingly; A previous subtract action for a texel may or may not become an addition action, depending on the value stored and the value added by the performed action.

3.5.3 Summary

This new tool, in combination with the two different layer types and adjustable opacities, greatly empowers the designer, making it possible to change the procedurally generated content, on both large and small scale, without worrying about losing any manual changes made if the terrain must undergo regeneration.

3.6 The River Tool

When a designer wants to add a river to the terrain, the River Tool (RT) can be utilized instead of manually carving the river into the terrain using the ST. The RT aims to be a convenient way to easily add not only simple rivers, but also entire river systems to the terrain. Figure 3.11 gives two examples, in the form of actual photographs, of what the goal of this tool is.

Figure 3.11: The goal of the RT is to make it possible for the designer to create rivers as shown in these examples. Photographs courtesy of images.google.com

3.6.1 Functionality

The basic idea of the RT is to have the designer place markers on the terrain which are then connected with bézier curves, thereby forming the path of a river. The markers are referred to as *river points*, and they hold specific information about the river at that particular point. For example, the river may be wider or deeper at one point than at another point of the river. For every pair of neighboring river points, the bézier curve between them is adjustable by a set of handles. These handles allow the designer to change the shape of the bézier going from one river point to the other. A river consists of minimum two river points, a beginning and an end. Figure 3.12 illustrates the use of handles, where the handles of the individual river points are adjusted to form a bend in the river.

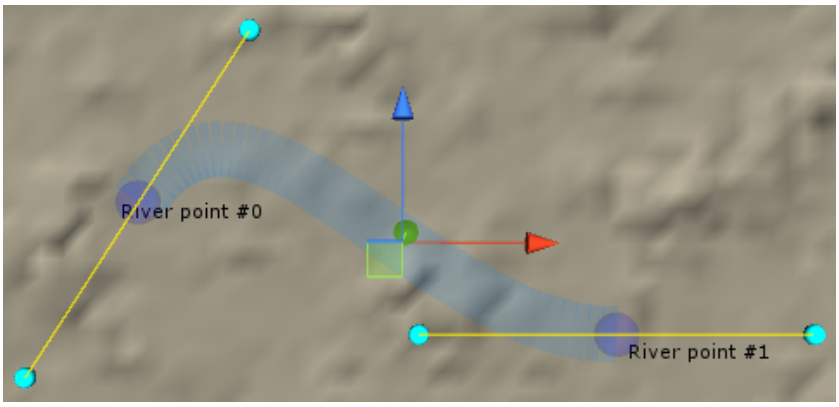


Figure 3.12: A basic river consisting of two river points. The colored gizmos help illustrate the final outcome of the river.

The river is displayed to the designer with a graphical overlay, where river points are labeled and shown as spheres, and the bézier between them is clearly visible. The designer can adjust

the handles (shown as teal spheres in Figure 3.12) to change the path of the river.

Additional river points can be added to any existing river point, which is demonstrated in Figure 3.13, where two new river points are added to one of the existing river points, causing the river to split into two. Furthermore, in Figure 3.13 the width of the river changes, which is attained by increasing the radius of the first river point. The width and depth of the river can be changed for any river point along the river.

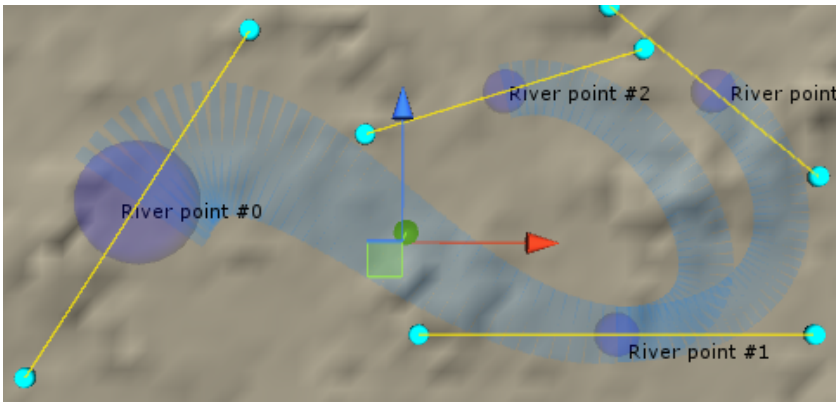


Figure 3.13: A forked river consisting of four river points, where the with of the river changes over distance.

3.6.2 Implementation

With the functionality for designing a river in place, we need to consider the limits of the tool and how it should act in different scenarios. The RT will carve a path in the existing terrain, which is a essentially a height field, which means we are limited to one elevation value for any point on the terrain. Most importantly, we need to decide how the tool will add a river to an existing terrain, a terrain which can have any number of ecosystems generating varying elevation values. One approach would be to completely reshape the terrain covered by a river path, ignoring some or all of the information gathered from the procedural system and layers. Another approach would be to simply lower the elevation values of the terrain points overlapped by a river path. The second approach was chosen as it allows for more flexibility, i.e. having layer content influence a river if so desired. The set of river points that make up a river each have a depth value associated with it, which can be used to find an interpolated depth at any point along the river. However, there is nothing to prevent the designer from overlapping or crossing a river with another river, and this is by design; we want to be able to make a network of intertwined rivers, as exemplified in Figure 3.11. But, when two or more rivers overlap, the depth for that particular point is decided by the deepest of the involved rivers. This is the solution chosen for height fields. Rivers for volumetric terrains would have to be handled differently, as any number of rivers could run at different elevations, e.g. a mountain stream on top of an underground cave river.

The designed river will subtract an interpolated value from the underlying terrain, and the design of the river must be capable of being reapplied to a regenerated terrain. We already have a method for storing and applying elevation changes, the Additive Layer, which is what rivers are saved as. The additive layer (Section 3.4.3) ignores other layers and subtracts the elevation values stored for the various terrain points. To convert a designed river into an additive layer, every segment of all rivers are processed to find the exact elevation change for every point on

the terrain. This processing task can take a while to complete initially, but once it is stored in a layer it is computationally cost free to reapply. However, if a river is manipulated, this process will have to be rerun.

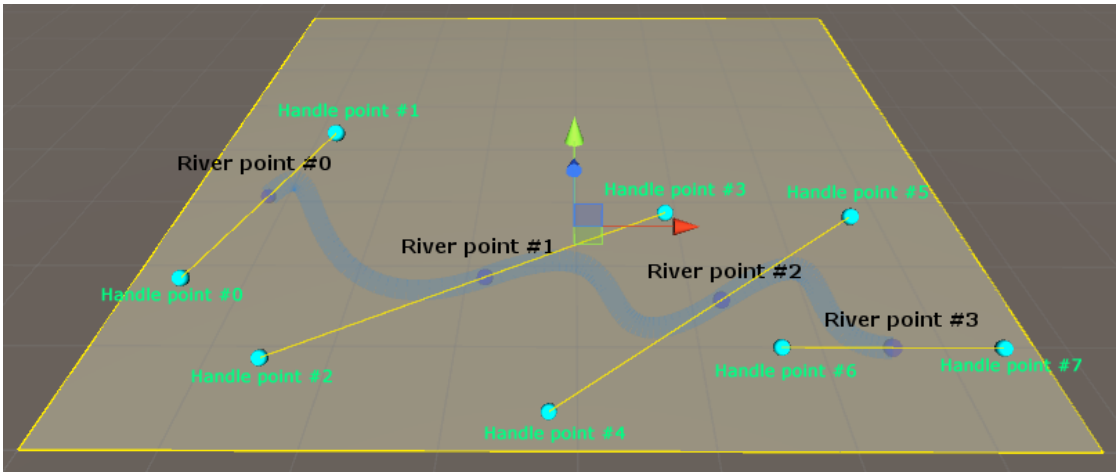


Figure 3.14: A river being designed with the RT by placing connected river points and manipulating the bézier curves between them.

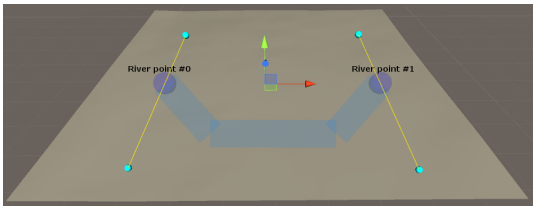
The entirety of the river is a composite bézier spline, made up of the bézier curves between each connected pair of river points. Every bézier curve is cubic, meaning the curve is made up of four control points: the two river points being connected, and the positions of the handles for two river points. Figure 3.14 illustrates a simple river made up of four river points, i.e. three bézier curves. The four control points, which make up each bézier curve of the composite bézier spline, are shown in Table 3.1. In the example given, there are two points not being utilized: Handle point #0 and Handle point #7. These two handles could have been hidden for clarity. In fact, it would have been enough with a single handle for each river point, as the vector for the incoming handle is directly opposite of the outgoing handle: $\vec{in} = -\vec{out}$. But, by having two handles, it becomes visually easier to tweak both the incoming and outgoing path of the river. Making the handles opposites of each other was a design choice, as it results in more natural-looking rivers, than if they were not.

	p1	p2	p3	p4
1st curve	River point #0	Handle point #1	Handle point #2	River point #1
2nd curve	River point #1	Handle point #3	Handle point #4	River point #2
3rd curve	River point #2	Handle point #5	Handle point #6	River point #3

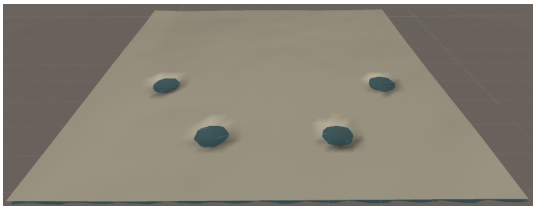
Table 3.1: The composite bézier spline is made up of three bézier curves, using the control points p1, p2, p3 and p4.

As with the many other aspects of terrain modeling, we want to have a smooth transition between the elements, in this case the ecosystem elevations and the river. Therefore, an interpolation curve is added on a per river basis, making it possible for the designer to adjust how a river transition to the underlying ecosystem. The interpolation curve is used for three calculations:

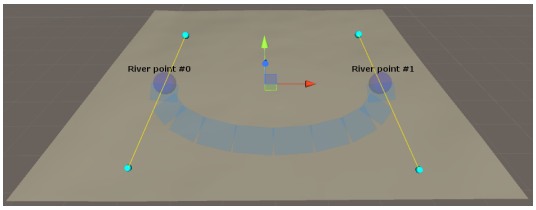
the depth and width between two river points and the depth from the rivers edge to its centerline. Several interpolation curves could be used, but a single was chosen for simplicity. Using a basic s-curve results in smooth river walls and a gradual change in elevation as the river progresses to the next river point. A default width and depth is set for a river, which can then be sampled or overruled by individual river points.



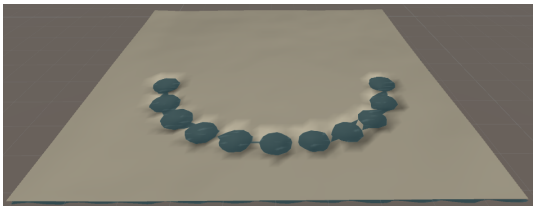
(a) Composite bézier spline with river quality: 3



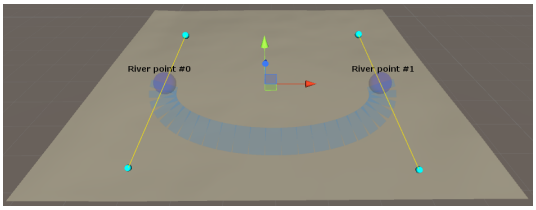
(b) Resulting river with a quality of 3



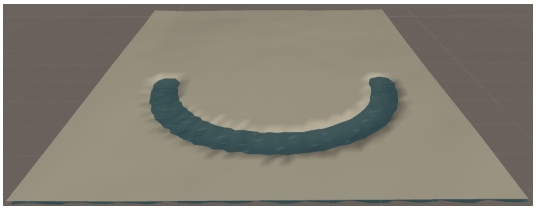
(c) Composite bézier spline with river quality: 10



(d) Resulting river with a quality of 10



(e) Composite bézier spline with river quality: 25



(f) Resulting river with a quality of 25

Figure 3.15: Different quality rivers, all using an s-curve for interpolating the depth and radius of individual points. The quality of a river determines the amount of sample points along its path.

To determine which points of the terrain are influenced by the constructed rivers, a *quality* parameter is introduced. The quality parameter specifies the amount of sampling points along the individual bézier curves of the river. Then, to determine if a terrain point should be influenced by the river we calculate the distances from that terrain point to every sampling points of the constructed rivers. If the distance between the terrain point and a sampling point is less than the river radius at that particular sampling point, it becomes a candidate point. The depth and radius of the river, at any point along its path, is determined by the radius and depth parameters of the two connected river points and the custom interpolation curve of the river. The candidate point with the shortest distance to a terrain point will determine the depth at that terrain point.

Let

- $B(p_1, p_2, p_3, p_4, t)$ determine the point along the cubic bézier curve constructed using control points p_1, p_2, p_3, and p_4, at curve parameter t, see Equation 3.6 [6].

- R be the set of rivers.
- $P(R_i)$ be the set of river points along the i'th river of R.
- $P_j(R_i)$ be the j'th river point of river R_i.
- $N(R)$ be the number of rivers.
- $N(R_i)$ be the number of river points in river R_i.
- Q_i be the *quality* parameter of river R_i, where $0 < Q_i \in \mathbb{N}$.
- $F_i(t)$ be the custom interpolation curve of river R_i, at curve parameter t, where $0 \leqslant F_i(t) \in \mathbb{R} \leqslant 1$ and $0 \leqslant t \in \mathbb{R} \leqslant 1$.
- $H(P_j(R_i))$ be the position of the handle for river point $P_j(R_i)$.
- $S(P_j(R_i))$ be the set of sampling points making up the path connecting river point $P_j(R_i)$ to river point $P_{j+1}(R_i)$, of quality Q_i, see Equation 3.7.
- $S'(R_i)$ be the set of sampling points making up the entire river R_i, running through all its river points $P(R_i)$, see Equation 3.8.
- S be the collection containing all sampling points, of all connected river points, of all rivers, see Equation 3.9.
- S_k be the k'th sampling point of set S.

$$B(p_1, p_2, p_3, p_4, t) = \{p_1, p_2, p_3, p_4 \in \mathbb{R}^2, 0 \leqslant t \in \mathbb{R} \leqslant 1 \mid (1-t)^3\, p_1 + 3(1-t)^2\, t\, p_2 + 3(1-t)\, t^2\, p_3 + t^3\, p_4\} \tag{3.6}$$

$$S(P_j(R_i)) = \{\bigcup_{n=0}^{Q_i} B(P_j(R_i),\ H(P_j(R_i)),\ H(P_{j+1}(R_i)),\ P_{j+1}(R_i),\ n/Q_i)\} \tag{3.7}$$

$$S'(R_i) = \{0 < Q_i \in \mathbb{N} \mid \bigcup_{n=1}^{N(R_i)-1} S(P_n(R_i))\} \tag{3.8}$$

$$S = \{\bigcup_{n=1}^{N(R)} S'(R_n)\} \tag{3.9}$$

While building S, additional information is stored for every sample point S_k:

- $R(S_k)$ is a reference to the river containing sample point S_k.
- $r(S_k)$ be the radius of the river at sampling point S_k, using custom interpolation curve $F_{R(S_k)}$.
- $d(S_k)$ be the depth of the river at sampling point S_k, using custom interpolation curve $F_{R(S_k)}$.

Then, to determine if a terrain point p should be influenced by any river, we first select the sample points which are within range of point p, which we call candidate points. Finally, we get the depth value of the candidate point with the shortest distance to p, to determine the depth at p.

Let

- $C_1(p)$ be the subset of sample points of S, called candidate points, where the distance between the sampling point S_k and position p is less than or equal to the *radius* at sample point S_k, see Equation 3.10.

- $C_2(p)$ be the closest sample point in the candidate set C_1 to terrain point p, see Equation 3.11.

- $N(C_1(p))$ be the number of candidate points at terrain point p.

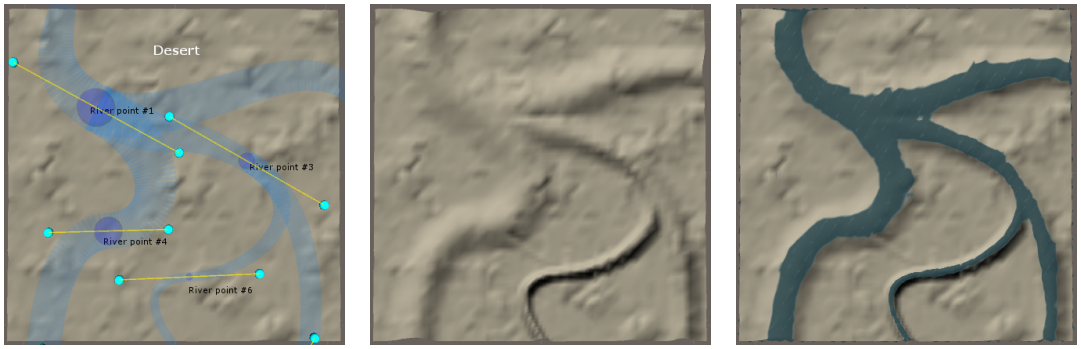
- $f_4(p)$ be the depth value at terrain point p, where the custom interpolation curve $F_{R(C_2(p))}$ of the river $R(C_2(p))$ and the radius $r(C_2(p))$ and depth $d(C_2(p))$ of the nearest candidate point $C_2(p)$ are used to determine the final result, see Equation 3.12.

$$C_1(p) = \{p \in \mathbb{R}^2 | \, ||p - q|| < r(q), \forall q \in S\} \tag{3.10}$$

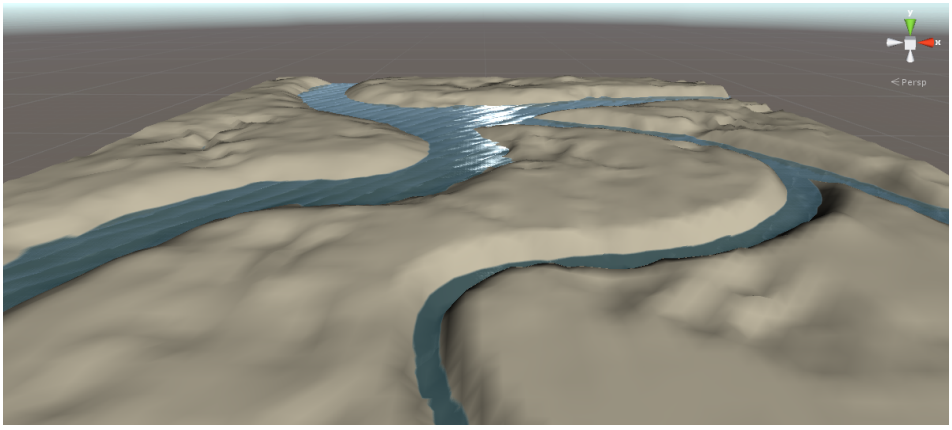
$$C_2(p) = \{\arg\min_{\forall q \in C(p)} ||p - q||\} \tag{3.11}$$

$$f_4(p) = \begin{cases} d(C_2(p)) \; (1 - F_{R(C_2(p))}(\frac{||p - C_2(p)||}{r(C_2(p))})), & \text{if } N(C_1(p)) > 0 \\ 0, & \text{if } N(C_1(p)) = 0 \end{cases} \tag{3.12}$$

Figure 3.15 demonstrate a simple river of varying quality using an s-curve for interpolating the depth and radius of the river. A complete example of a user designed river being applied to a terrain can be seen in Figure 3.16.



(a) User designed river. (b) Elevation values subtracted. (c) Waterplane added.



(d) The resulting river, with a custom waterplane added.

Figure 3.16: A concrete example of a designed river being applied to a desert terrain.

3.6.3 Summary

There are several benefits to using the RT over the ST for adding rivers. Visually, it is very easy to predict how a bézier spline is transformed into a river. And, it is significantly easier to tweak a set of bézier curves rather than changing raw elevation values stored in a layer. Terrain modeling is not a linear process, and changes will inevitably be required to suit new design specifications. The RT makes this a breeze, but, it is also important to realize its shortcomings. The RT does not handle creation of the actual rivers, but rather the path they will carve in the terrain. This means that rivers not deep enough to go below sea level will simply be dry riverbeds on the terrain. Likewise, it is not currently possible to have stream run downhill from a mountain unless the flowing river is somehow added manually. As the depth and shape of the river is stored in a hidden layer, it would be possible, as a future improvement, to construct a mesh from this information to function as a river. The current ST is best suited for relatively flat and steady elevation values, as shown with the desert ecosystem. However, as another improvement, every river point should have a lower final elevation value than its parent. Otherwise we end up with an unnatural river capable of flowing in multiple directions. Optimally, we want the river to flow downward toward the sea.

In conclusion, the RT, however limited, greatly contribute to giving control back to the designer.

3.7 Framework Feature Demonstration

The newly added framework functionality does not only let the designer shape the terrain after the procedural generation, it also enabled the designer to direct how the procedural processes are queried to form the terrain. To help demonstrate these new framework features, I now present a step-by-step example of how a virtual terrain could be created. As the new features only deal with elevation values, so will this example; terrain coloring and foliage placement will not be included. The aim of this example is to mimic the features of the landscape shown in the reference photograph, Figure 3.17.

Figure 3.17: The reference photograph used in the following example.

Step 1. Create initial Voronoi diagram

Looking at the reference picture, it is clear I need some very tall mountains, which gradually decent down to a river. To construct the initial terrain, I add the Voronoi terrain to the Unity scene and start placing ecosystems. Figure 3.18 is the initial Voronoi diagram constructed by placing ecosystems. The ecosystems are placed where occurring terrains features take place in the reference photograph; the *Tall Mountains* ecosystem are used where the snow capped mountains appear in the photograph. By placing the *Flatlands* ecosystem where the river is located, I will automatically get the gradual decent from the mountains, when the terrain is procedurally generated. With this simple layout, I get the procedurally generated terrain shown in Figure 3.19. The terrain is generated using IDW interpolation with an ordinary s-curve.

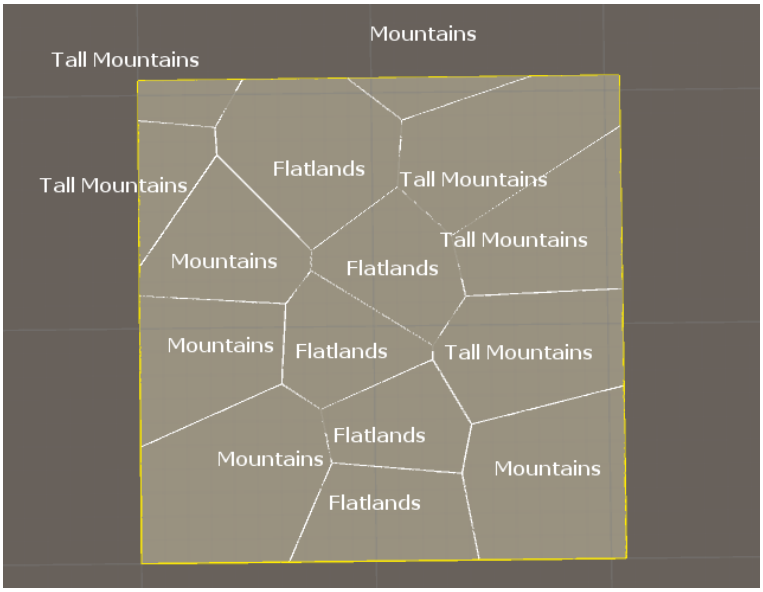


Figure 3.18: The initial Voronoi diagram designed to copy the layout of the terrain shown in the reference photograph.

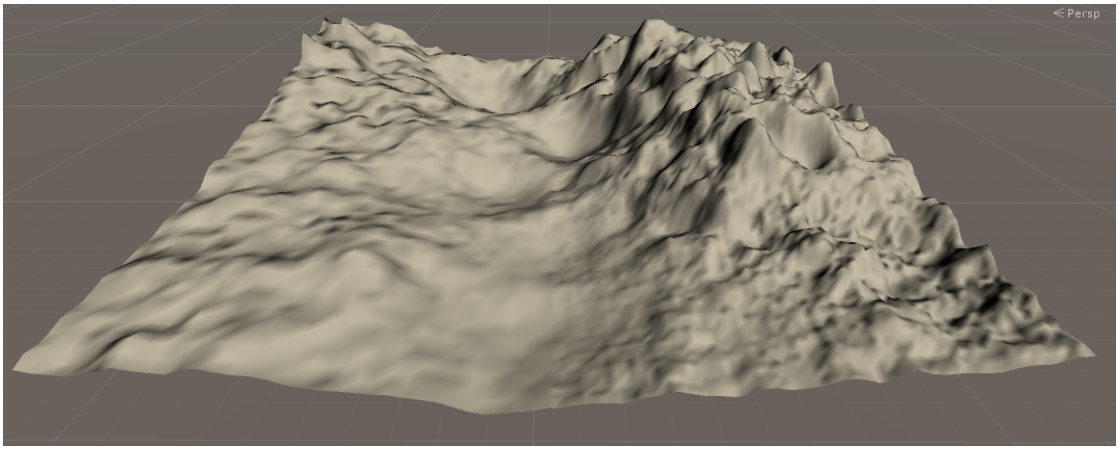


Figure 3.19: The procedurally generated terrain, based on the designed Voronoi diagram.

Step 2. Adjustments

The result is a bit more jagged than what is wanted, so to fix this I simply adjust the *Global noise scalar*, introduced in [11, Section 3.3.4], from a default value of 1, to 0.4. Secondly, the different ecosystems looks to be generally too influential; they are all blended together and their individual appearances become blurred. To fix this blending, I change the *power parameter* from 2 to 3, making the individual ecosystems more influential in their immediate vicinity.

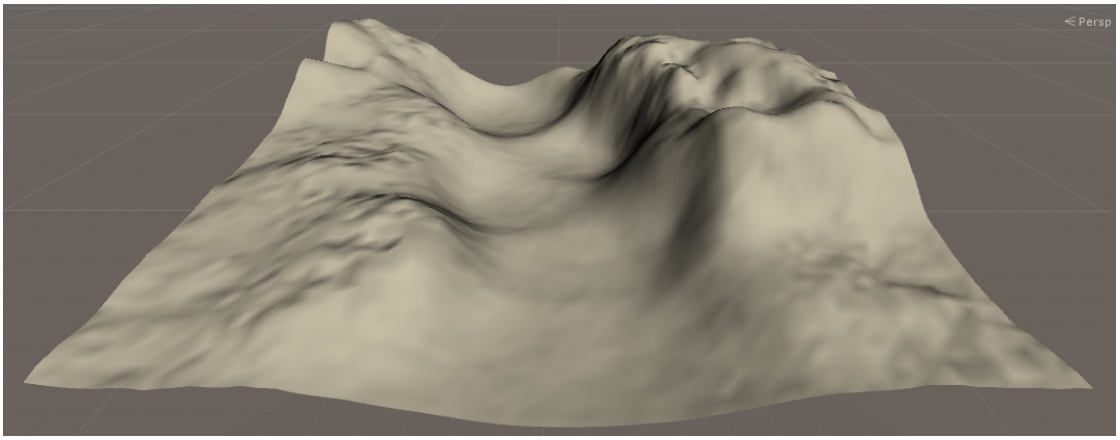


Figure 3.20: The result of adjusting a few parameters of the terrain.

Step 3. Adding the Waterplane

At this point, it would really be helpful to see the water in the scene, to get a better grasp of the scale of the terrain. so, I enable the waterplane and set it to a suiting elevation, as shown in Figure 3.21.

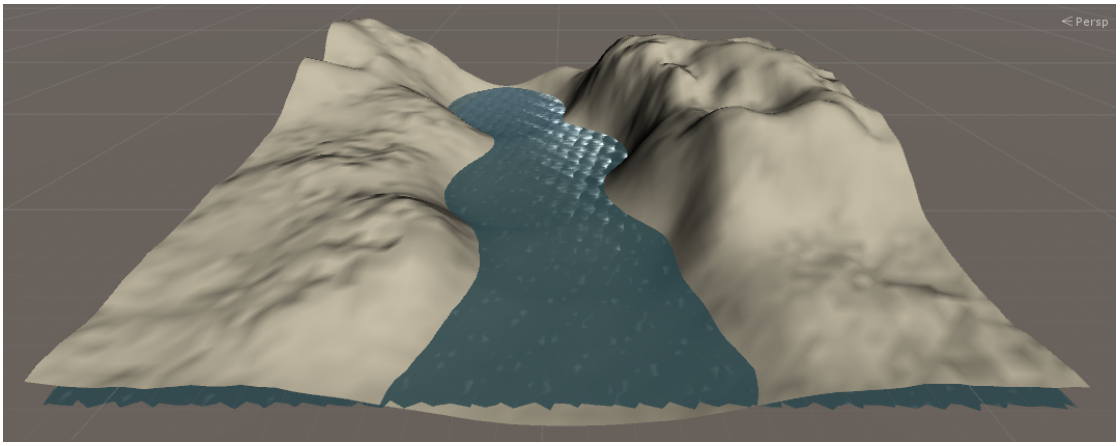


Figure 3.21: A waterplane is added to the terrain.

Step 4. Adjustments

With the waterplane in place, it becomes visually easier to tell that the mountains should actually be a lot taller. To allow for very tall mountains, I change the *y-scale* of the terrain from 40 to 80. To avoid the mountains being too steep, the *Tall Mountains* ecosystems are pulled back some to increase the distance to the river. At the same time, the elevation of the waterplane is lowered a bit, as the river is otherwise too wide. The results can be seen in Figure 3.22.

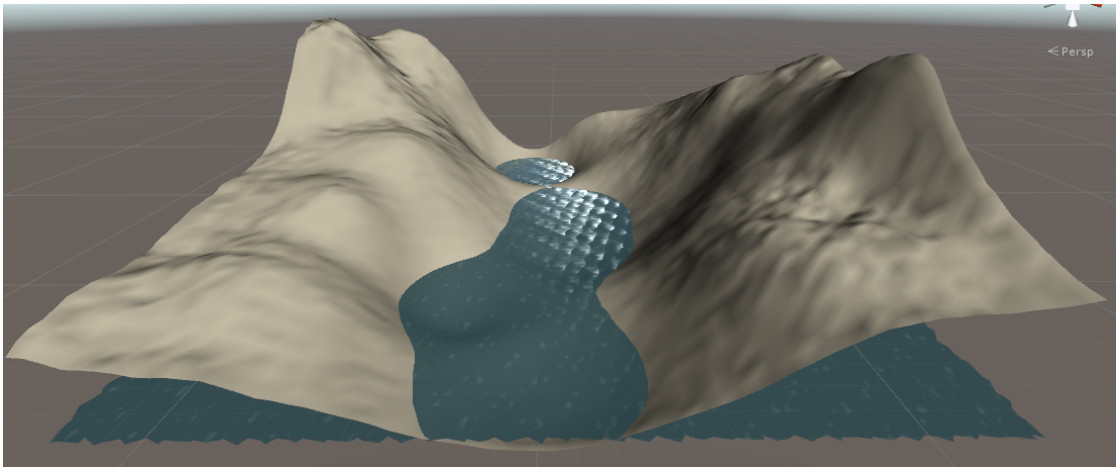


Figure 3.22: The new result reflecting adjustments to the Voronoi diagram and terrain scale.

Step 5. Sculpting

Along the river, in the reference photograph, are some flat areas. To implement these, I could relocate an existing, or add another, *Flatlands* ecosystem, however, I could also use the *Sculpting Tool*; I choose to demonstrate the latter. I add a new layer and start using the tool to decrease various parts of the terrain, shown in Figure 3.23.

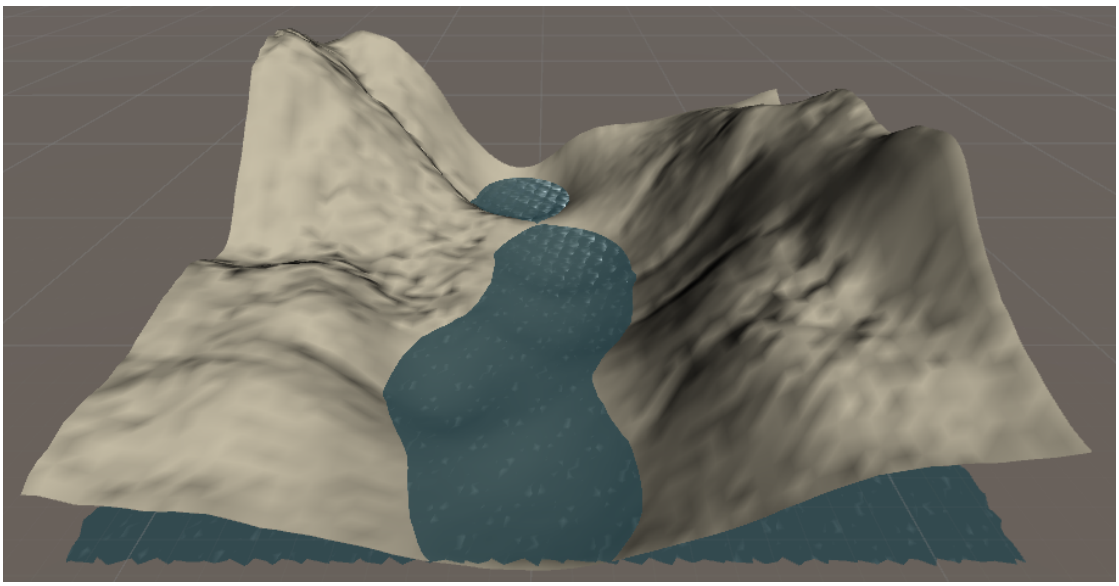


Figure 3.23: Using the sculpting tool to flatten and add detail to certain areas of the terrain.

Step 6. River details

The river is not very accurate to the reference photograph at the moment, so I want to try and fix that. I could continue using the sculpting tool to alter the slopes near the river, or I could use the *River Tool* to automatically decrement the elevation value on the path I specify, again, I choose to demonstrate the latter. The river added is shown in Figure 3.24.

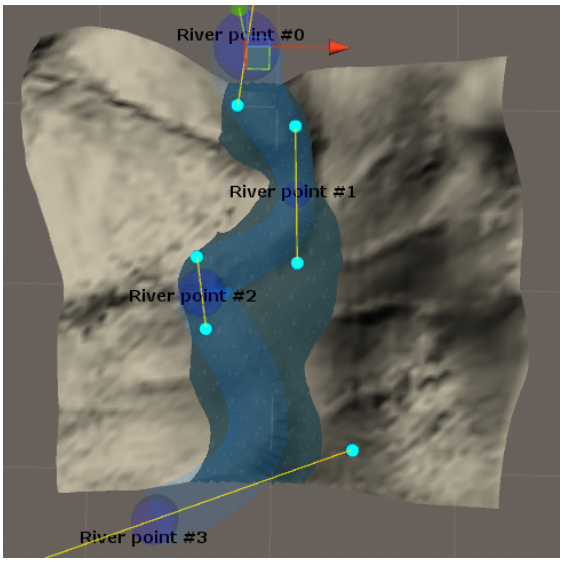


Figure 3.24: Using the river tool to lower the elevation of certain areas of the terrain.

Step 7. Results

This was a quick and dirty attempt at recreating real world terrain features on a virtual terrain, which at this stage can be seen in Figure 3.25. As is clearly evident, the terrain creation process is iterative, and many additional steps would be required to finalize the example terrain. Another demonstration of mimicking natural rivers with the river tool is exemplified in Figure 3.26.

The humongous task of creating a terrain intended for computer games, simulators and other virtual worlds, is normally very time intensive, but, can be significantly lessened by using a procedural framework, such as the one proposed herein.

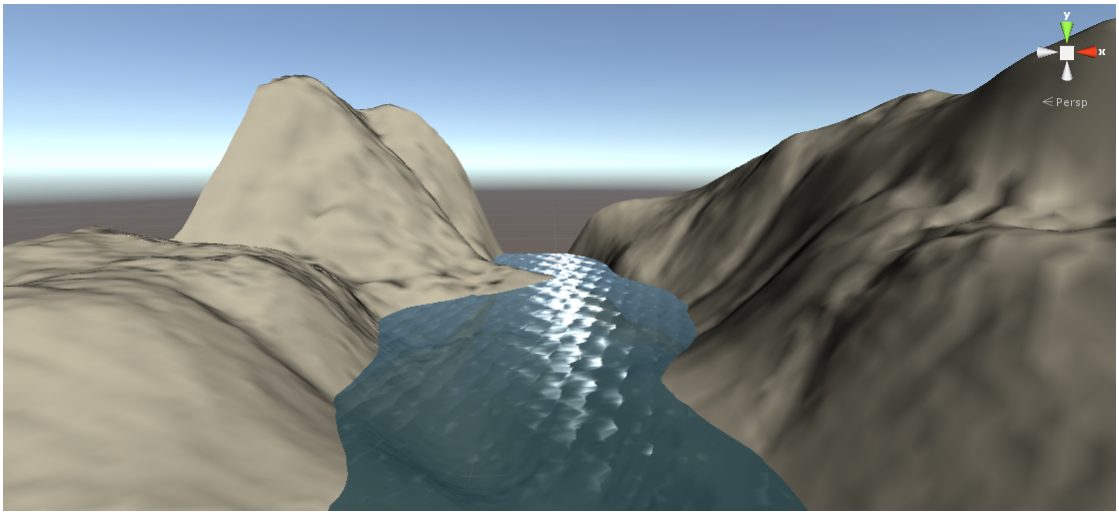


Figure 3.25: The resulting terrain, constructed in a few steps using the procedural framework.

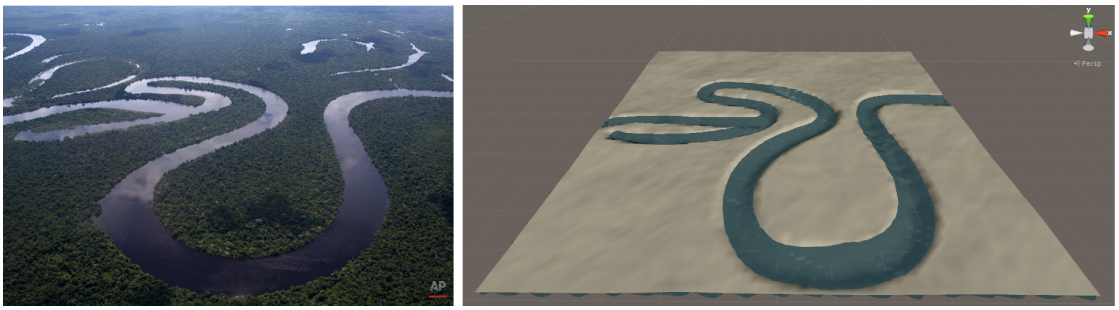


Figure 3.26: An example of using the river tool to copy the features of actual rivers.

4 Conclusion

PM have great potential when used in the construction process of virtual terrains, where it can mimic many natural terrain features, and thereby significantly reduce the time spent on manual sculpting of the terrain. However, setting up a PM system and defining the outcome of these require expertise and can be a cumbersome task in itself, involving a lot of trial and error testing. The difficulty of accurately guessing the outcome of these procedures have kept designers from adopting the procedural approach, and instead rely heavily on manual content creation. The framework prototype, as described in this report, overcome these problems by introducing designer-friendly methods for defining and manipulating the terrain being procedurally generated in order to benefit from both the speed of procedural modeling and the precision of manual content creation. A rough terrain is created by visually defining where specific ecosystems should occur. This is a quick method for creating the basis of the terrain, which can then undergo manual adjustment to reach the wanted result. Two custom tools are implemented to help ease refinement of the procedurally generated terrain: the sculpting tool, which is used for simple sculpting, and, the river tool, which uses visual cues to carve out rivers in the terrain.

How can artistic control be retained?

In the analysis chapter, both basic and advanced tools for terrain manipulation were analyzed. The brush (Section 2.1) being the simplest and most natural tool to use is a must-have for any graphics application. This led to the sculpting tool being implemented in Section 3.5, with adjustable brush size and strength. With the addition of the sculpting tool it became possible to perform instant changes, in the form of elevation addition and subtractions, on the procedural terrain. The river tool implemented in Section 3.6 is partly based on the features curves examined in Section 2.3, but kept simpler to avoid making maintenance of rivers too complex. Certain characteristics, such as width and depth, are set on a per river basis, which individual points along the river can be set to override, if so desired by the designer. These tools help manipulate the procedurally generated terrain, whereas the new approach to handling ecosystems, described in Section 3.3, help steer the procedural process. With the manual placement of ecosystems and the visual Voronoi overlay, the resulting terrain becomes easier to foresee. Collectively, the two additions to the base prototype, contribute to retaining the artistic control in a procedural modeling framework.

How can manual changes be preserved and survive the regeneration process?

Inspired by both Photoshop [1] and SketchaWorld (Section 2.4), layers were introduced to serve primarily one purpose: storing information about manual changes performed on the procedural terrain. However, it quickly became evident that they could also serve to help attain artistic control. Two different types of layers were implemented and the visibility of individual layers

was made manipulative. Layers was implemented in Section 3.4 and provides the framework with an effective method for storing, and eventually reapplying, manual changes performed on the procedural terrain. The layer functionality was also utilized to store the elevation changes performed with the river tool. Layers are the means to preserve manual changes, but they are not limited to this one task. There are many ways to expand the layer functionality, as are explained in Section 5, Future Work.

Procedural modeling, of virtual terrains intended for computer games, becomes a viable alternative to manual creation when designer-friendly methods, for performing changes on the procedurally generated terrain, are introduced.

The previous demonstration (Section 3.7) showed that it was possible to construct a virtual replication of an actual terrain in only a few steps. The initial terrain, based on the Voronoi diagram, was created in minutes, and could quickly and easily be rearranged. With the procedural terrain, it was possible to make changes with the custom tools to more accurately portrait the features shown in the photograph.

Finally, do these features make the procedural framework a viable alternative to manual creation of terrains? Yes, such features makes the procedural approach highly desirable. With these features, the framework becomes a great asset for the game designer, which help expedite the terrain creation process greatly.

5 Future Work

The future work chapter elaborate on possible framework expansions and improvements, which could be used in a future version of the framework.

5.1 General Improvements

The various feature implementations have been completed with minor computational optimization, such as reusing previously generated procedural values, if the procedural system is unchanged. Likewise for rivers, if their placement or parameters remain unchanged there is no need to recalculate their outcome. However, all calculations are currently done on the CPU, which can be slow for large terrains. Utilizing the parallelism of the GPU could potentially speed up some of these calculations.

Due to some heavy computational requirements in the current framework prototype, some of the features are not reflected in real-time, but rather left for the designer to manually update by clicking a button. The optimal solution would be to reflect any and all changes whenever they happen.

5.2 Tools

As with any trade, you do not have a single tool to fulfill all purposes, but a set of tools, each specialized to tackle some task. The two implemented tools, the ST and RT, help the designer sculpt the procedurally generated terrain in their own way, but, there are may other specialized tools the framework would benefit from including.

The Smoothing Tool

The purpose of the smoothing tools is to smooth out an area, by using some filter, e.g. a Gaussian blur, over an area. The tool could be implemented as a brush, where size and smoothing strength could be adjusted by the designer. This tool would be particularly helpful in combination with other sculpting tools, to lessen their transitions; like sandpaper for a carpenter.

The smoothing tool would also be useful for flattening an area, which is often wanted when placing man-made objects on the terrain.

The Eraser

The eraser should acts like the ST but with opposite effect, instead of adding/subtracting elevation values, it will disable affected texels or decrease their effectiveness be some degree. Currently, it is not possible to remove affected texels on a layer, meaning once a texel have been manipulated it will always overlap texels in subsequent layers.

The Freezing Tool

When manipulating the terrain, the designer makes changes based on the elevation values generated by the procedural system. If the procedures are changed and the terrain undergoes regeneration, the changes performed by the designer may no longer suit the generated terrain.

Therefore, it would be useful to be able to define areas which freezes the underlaying elevation values and ignores any new elevation values produced by a change in the procedural system. The two different elevation values would have to gradually converge at the boundaries of the frozen areas. The designers terrain customization would then still be valid even after the procedural system have been changed.

Improving the Sculpting Tool

The ST could be improved by enabling additional brush patterns as shown in Figure 2.1, or even user defined patterns. It would also be handy to be able to sample elevation heights on the existing terrain. The sampled value could then be used as the strength of the ST, and terrain manipulation could be sped up, by avoiding parameter tweaking.

Improving the River Tool

As already mentioned in Section 3.6.3, the RT does not facilitate actual river creation; the designed rivers are used to carve out the path of the river in the terrain, by subtracting elevation values. A massive improvement to the RT would be to add this missing functionality. The actual river could be constructed from the information already stored in the designed river.

Another improvement would be to ensure that the river always flow downward, forced by gravity. Currently, there is nothing to prevent the carved river from flowing up and down hillsides, which could become a visual problem.

5.3 Layers

The elevation changes performed by the designer are stored in layers, whose size is based on the predefined size of the terrain. however, as it is possible to change the size and scale of the terrain at any time, the information stored in the layers should scale appropriately, or alternatively relocate to the center of the resized layer.

A very usable feature for layers would be designer directed rotation and scaling of the layer content. And, like in Photoshop [1], make it possible to merge multiple layer into a single layer.

Another improvement would be to visually show the content of individual layers, to let the designer know where on the terrain the different layers occupy.

Bibliography

[1] Adobe: Adobe Photoshop. – http://www.adobe.com/products/photoshop.html

[2] Beutel, Alex ; Mølhave, Thomas ; Agarwal, Pankaj K.: Natural neighbor interpolation based grid DEM construction using a GPU. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems ACM (Veranst.), 2010, S. 172–181

[3] Carpentier, Giliam J. de ; Bidarra, Rafael: Interactive GPU-based procedural heightfield brushes. In: Proceedings of the 4th International Conference on Foundations of Digital Games ACM (Veranst.), 2009, S. 55–62

[4] Gain, James ; Marais, Patrick ; Strasser, Wolfgang: Terrain sketching. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games ACM (Veranst.), 2009, S. 31–38

[5] Hnaidi, Houssam ; Guérin, Eric ; Akkouche, Samir ; Peytavie, Adrien ; Galin, Eric: Feature based terrain generation using diffusion equation. In: Computer Graphics Forum Bd. 29 Wiley Online Library (Veranst.), 2010, S. 2179–2186

[6] Riškus, Aleksas: Approximation of a cubic Bézier curve by circular arcs and vice versa. In: Information technology and control 35 (2015), Nr. 4

[7] Shepard, Donald: A two-dimensional interpolation function for irregularly-spaced data. In: Proceedings of the 1968 23rd ACM national conference ACM (Veranst.), 1968, S. 517–524

[8] Smelik, Ruben M. ; Tutenel, Tim ; Kraker, Klaas J. de ; Bidarra, Rafael: A proposal for a procedural terrain modelling framework. In: Poster Proceedings of the 14th Eurographics Symposium on Virtual Environments EGVE08, 2008, S. 39–42

[9] Smelik, Ruben M. ; Tutenel, Tim ; Kraker, Klaas J. de ; Bidarra, Rafael: A declarative approach to procedural modeling of virtual worlds. In: Computers & Graphics 35 (2011), Nr. 2, S. 352–363

[10] Technologies, Unity: Unity. 2016. – http://unity3d.com/

[11] Wivelsted, Toke A.: A Framework for Sketch-Based Procedural Modeling of Terrains for Use in Computer Games. 2016

A Abbreviations

The following table lists the abbreviations used for long and technical words.

PM	Procedural Modeling
GPU	Graphical Processing Unit
NNI	Natural Neighbor Interpolation
IDW	Inverse Distance Weighting
NeNI	Nearest Neighbor Interpolation
ST	Sculpting Tool
RT	River Tool

B Nearest Neighbor Interpolation

NeNI is a simple and fast method for interpolation. For any point on the terrain, the ecosystem closest to that point is the sole contributor. Using NeNI on the Voronoi diagram from Figure 3.3 gives the result shown in Figure B.1. NeNI has a major flaw: the transition between different ecosystems is sharp and very noticeable, resulting in unrealistic terrain features.

To improve the NeNI approach, the various Voronoi cells need to converge their elevation values with neighboring cells. However, we want to avoid looking at more than one Voronoi cell when using NeNI, so a solution is to interpolate between the elevation value given by the ecosystem and some set elevation value, referred to as the *neutral value*. The closer a point on the terrain is to the center og the Voronoi cell, the greater it is influenced by the ecosystem elevation. Terrain points near Voronoi cell edges should only be influenced by the neutral value.

For every terrain point x, we need to determine the length of the vector going from the Voronoi cell origin c (the ecosystem point), going through x and eventually hitting either a cell edge or the terrain bounds d, i.e. d is the intersection point. Then, we can calculate the final elevation value of a terrain point by interpolating between the ecosystem elevation and neutral value as such:

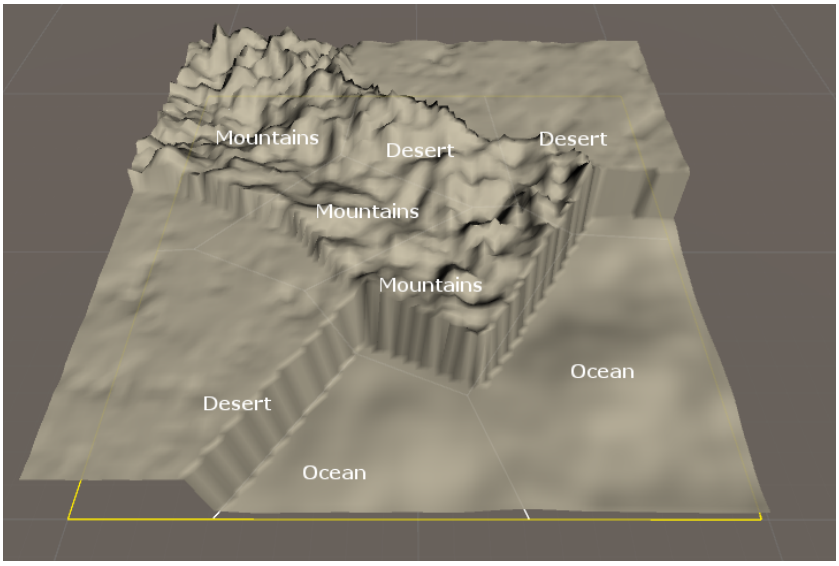


Figure B.1: Nearest Neighbor Interpolation, illustrating the sharp transitions between ecosystems.

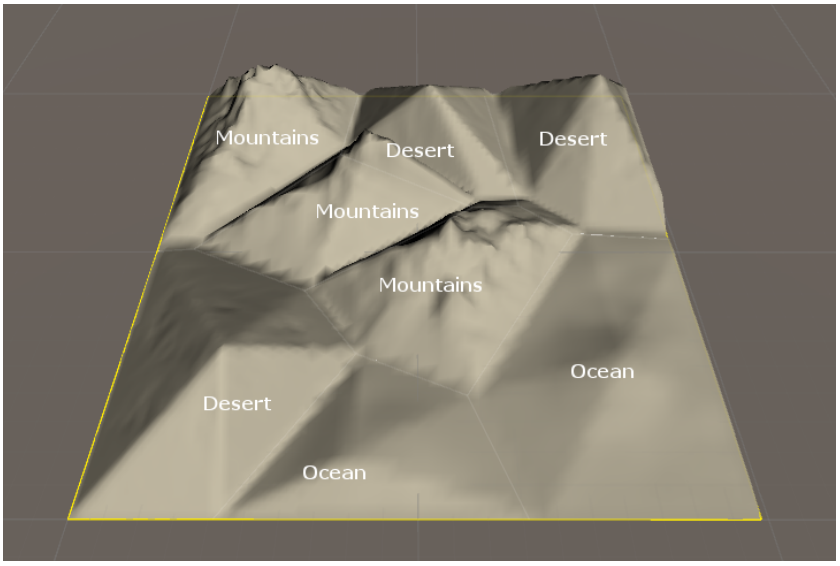


Figure B.2: Nearest Neighbor Interpolation, using linear drop-off.

$$\alpha = \frac{|x - c|}{|d - c|}$$
$$elevation(x) = e * (1 - \alpha) + n * \alpha,$$

where e is the ecosystem elevation and n is the neutral value. This converging effect can be seen in Figure B.2, where linear interpolation is used to determine the result.

As linear interpolation causes sharp edges, an adjustable interpolation curve is added. The user can chose to between some predefined curves like linear and s-curve, or, design his own by placing nodes on the curve and tweaking their properties. The curve is implemented in the Unity editor for the terrain and can be seen in Figure B.3 where the resulting terrain is also shown.

In the previous examples, the neutral value have been set to zero, i.e. the lowest possible elevation. But, rather than converging on a preset value, it is also possible to converge on an ecosystem instead. The neutral value is dynamically set for every point of the terrain, querying a specified default ecosystem elevation. This approach can result in a completely different terrain, often resulting in better ecosystem transitions, as shown in Figure B.4.

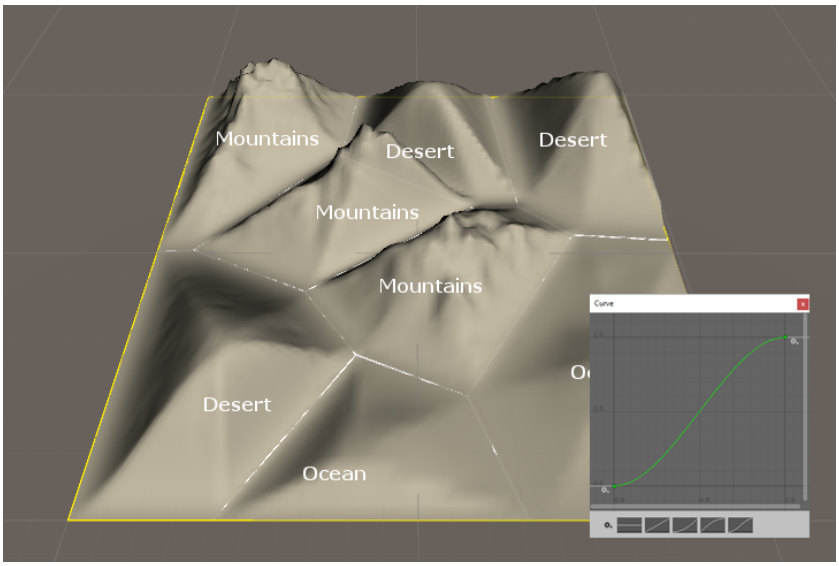


Figure B.3: Nearest Neighbor Interpolation, using an s-curve.

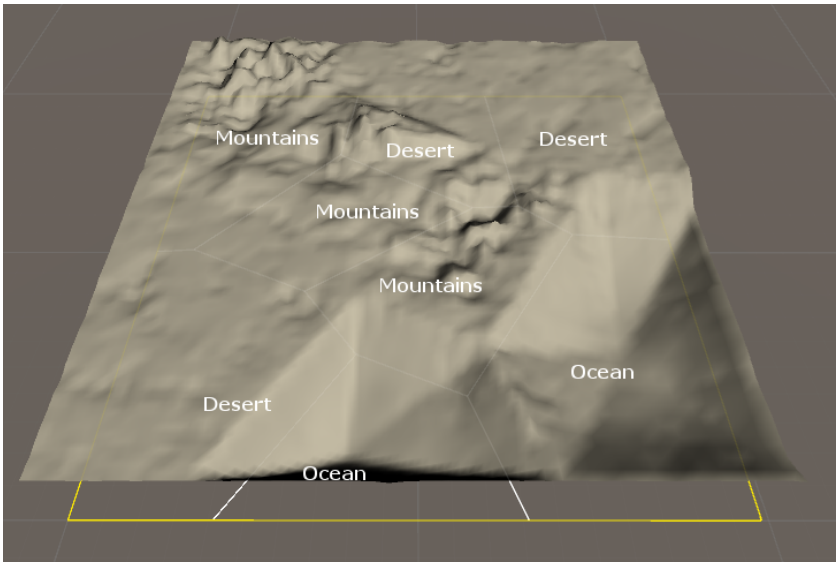


Figure B.4: Nearest Neighbor Interpolation, where the neutral value is fetched from the Desert ecosystem.

C Default Ecosystem Elevation Samples

The following images show a sample of the elevation values returned when querying the different ecosystems, and can be used for reference when reading about the various interpolation methods.

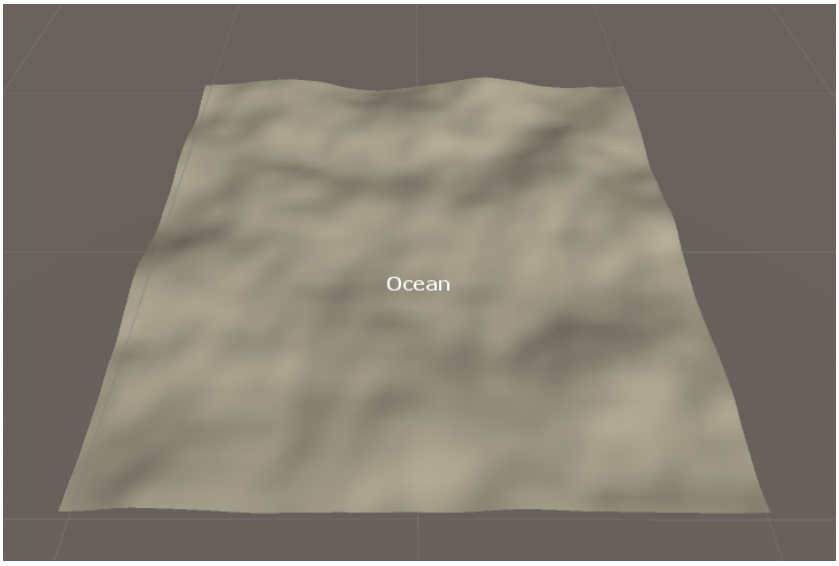


Figure C.1: The ocean ecosystem; smooth rolling hills with low frequency and low elevation values.

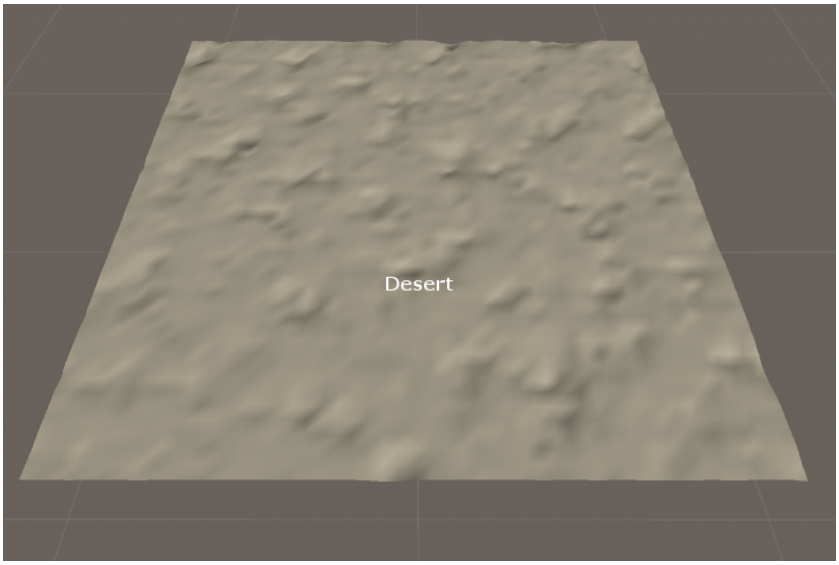


Figure C.2: The desert ecosystem; an almost flat terrain with compressed elevation values in the mid-range.

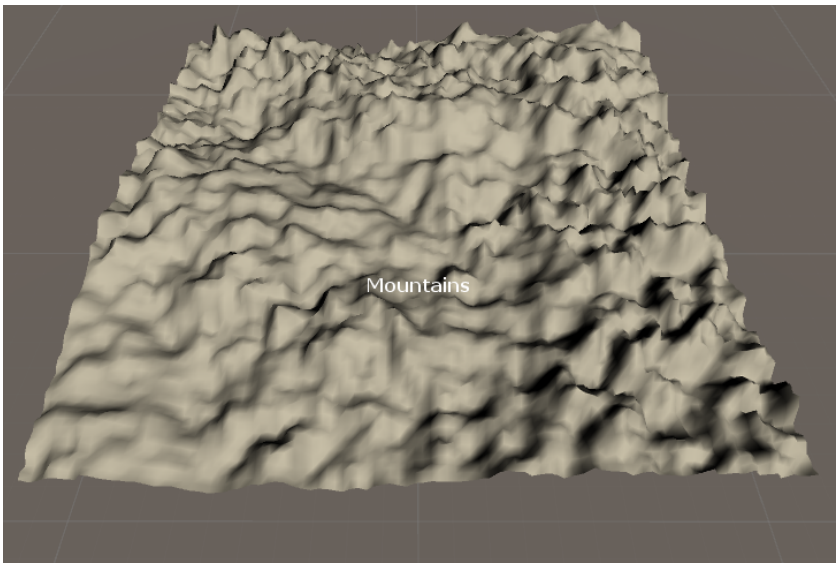


Figure C.3: The mountains ecosystem; high frequency jagged mountains with elevation values going from low to high.

D Installing and Using the Prototype

The latest version of Unity can be downloaded at **http://unity3d.com/** and the framework prototype can be accompanying the report, in the folder named "**Prototype**\". The prototype was built and tested in Unity 5.3, but should also work in newer versions. Some elements from the standard package of Unity are used, which is why they are included with the prototype.

Once Unity is up and running, the prototype is opened as any other Unity project, by going to "File" → "Open Project...". When the project have loaded fully, a terrain generated by the prototype should be visible in the "Scene" view. If this is not the case, the test-scene containing the generated terrain can be opened by going to "File" → "Open Scene..." and navigating to the folder "\Prototype\Assets\" and selecting the "Scene.unity" file. The prototype does not require the test-scene to be active for it to function, the test-scene is merely meant to provide a default environment to help the user get situated.

A new Voronoi terrain object is added to the scene through the menu: "GameObject" → "3D GameObject" → "Custom Voronoi Terrain". Once the menu-item "Custom Voronoi Terrain" has been selected, the new terrain is added to the scene. Attached to this gameobject are four components: *Terrain Settings*, *Terrain Tools*, *Terrain Layers*, and *Voronoi Terrain*.

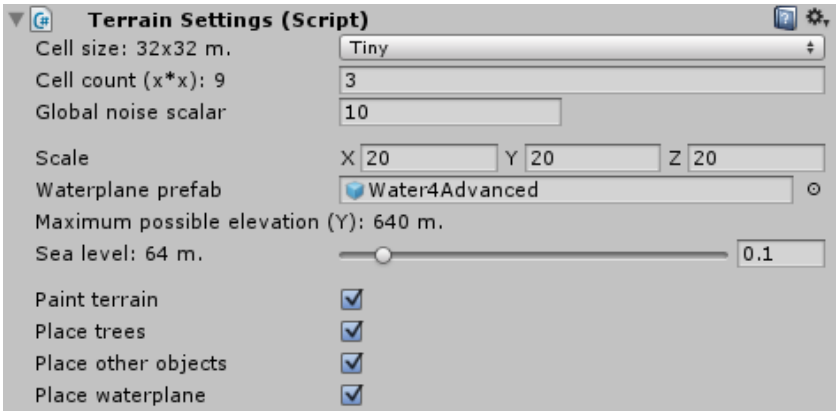


Figure D.1: The *Terrain Settings*-component.

The *Terrain Settings*, shown in Figure D.1:

Cell size specifies the size of the individual terrain segments. The larger these are, the longer it will take to process them.

Cell count specifies the amount of segments (input squared) the entire terrain consists of.

Global noise scalar is a multiplier parameter used to scale all sampled noise functions.

Scale is a multiplier parameter to scale the terrain.

Waterplane prefab specifies the gameobject to function as the waterplane of the terrain.

Sea level specifies the elevation of the waterplane.

Paint terrain is non-functional in current prototype.

Place trees is non-functional in current prototype.

Paint other objects is non-functional in current prototype.

Place waterplane specifies whether the Waterplane prefab is instantiated when creating the terrain.

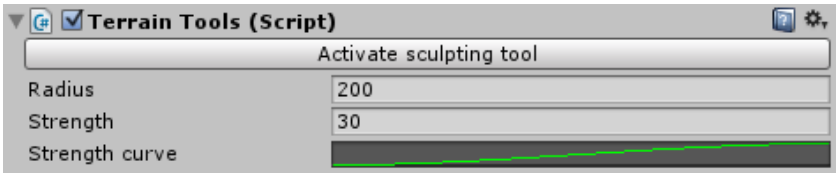


Figure D.2: The *Terrain Tools*-component.

The *Terrain Tools*, shown in Figure D.2:

Radius specifies the radius of the brush.

Strength specifies the power of a single mouse-click, i.e. how much the terrain is raised or lowered.

Strength curve is a customizable curve that specifies the brush's opacity, as a function of the radius to the center of the brush.

When the sculpting tool is active, a green disc is shown when hovering the mouse pointer over the terrain, illustrating the area of effect. Left-clicking with the mouse will raise the terrain whereas holding ctrl and left-clicking will lower the terrain.

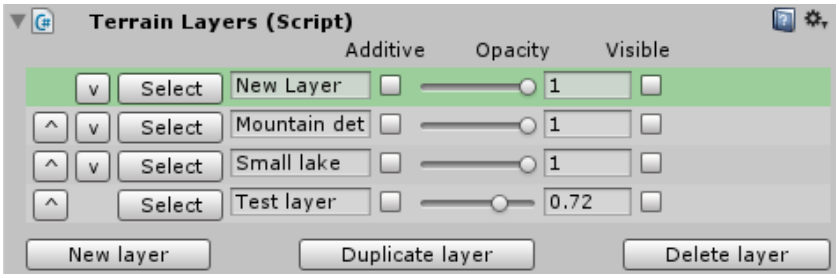


Figure D.3: The *Terrain Layers*-component.

The *Terrain Layers*, shown in Figure D.3 contain the complete set of layers for the terrain. The set of layers are displayed in a list, where order is determined from top to bottom. Layer

order can be altered by using the left-most buttons. A new layer can be added by clicking the "New layer"-button. Actions performed with the different tools are only applied to the active layer, highlighted in green. To toggle which layer is the active layer, simply press the "Select"-button. A layer can be set as *additive* by toggling the check-box. The opacity of the layer is set by adjusting the widget named *opacity*, clamped to [0;1]. Individual layers can also be disabled entirely by toggling the *Visible* check-box. In Figure D.3, all layers are disabled. Layers can also be named for convenience. The "Duplicate layer"-button and "Delete layer"-button are non-functional in the current prototype.

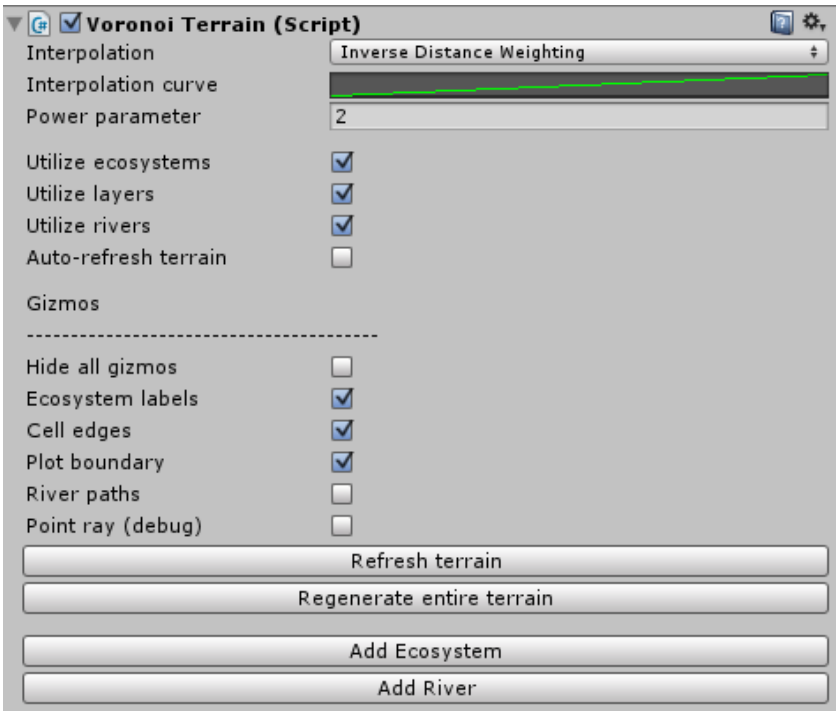


Figure D.4: The *Voronoi Terrain*-component.

The *Voronoi Terrain*, shown in Figure D.4:

Interpolation specifies the method for interpolating the designer-placed ecosystems.

Interpolation curve is a customizable curve that specifies the curve used for the IDW method.

Power parameter, also used for IDW, specifies how influential placed ecosystems should be. A low power parameter causes ecosystems to blend together, where a high power parameter has the opposite effect.

Utilize ecosystems toggles whether the placed ecosystems are used for determining the elevation values of the terrain.

Utilize layers toggles whether the layers are used for determining the elevation values of the terrain.

Utilize rivers toggles whether the designed rivers are used for determining the elevation values of the terrain.

Gizmos can be toggled on to help illustrate various information (labels of the placed ecosystems, the path of created rivers etc.) on the terrain, or toggled off to hide the same information.

The "Refresh terrain"-button is intended to update the elevation values of the terrain without reconstructing the segments that contain no changes. This is however not entirely functional in the current prototype. Instead, the "Regenerate entire terrain"-button should be used, which is the button that initializes the calculation of elevation values for the entire terrain. Clicking this button will stall Unity for a while, depending on the complexity chosen interpolation method and parameter values. Once the process has completed, the terrain is automatically updated to show the new elevation values.

Generally, if a change has been made to the terrain and is not immediately reflected, the "Regenerate entire terrain"-button will have to be clicked.

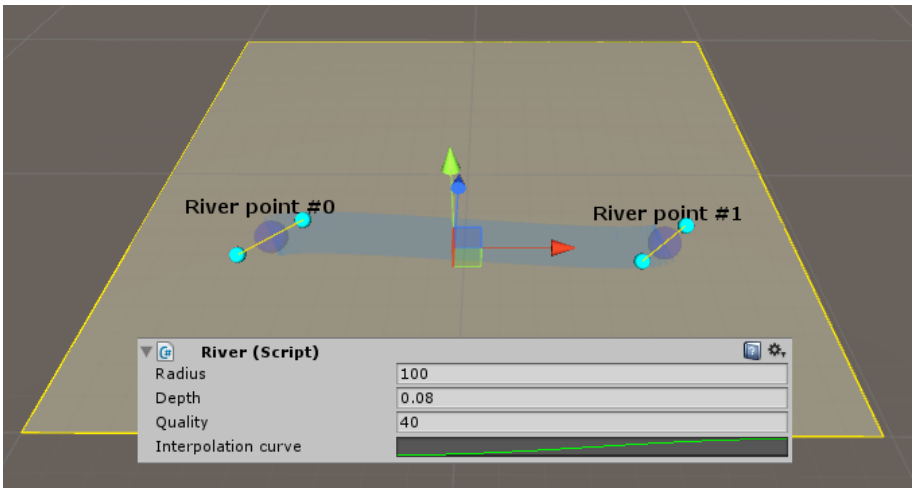


Figure D.5: A default *River* is added to the terrain.

The "Add ecosystem"-button placed a reference to an ecosystem on the terrain, in the form of a Unity gameobject. The ecosystem-gameobject can be freely moved by the designer, and its placement determine the generated terrain.

The "Add-river"-button constructs and places a simple river on the terrain, referred to as the river tool in the report, demonstrated in Figure D.5. The river contains two river-points by default, which can be moved around by the designer. A river contains four adjustable parameters:

Radius specifies the default radius of the river, i.e. the width of the river is $2 * radius$.

Depth specifies the default depth of the river, clamped to [0;1].

Quality specifies the number of sample-points between each pair of river points. A higher quality river yields a better looking end result.

Interpolation curve specifies the curve used for interpolating the depth and radius of the sample-points between river-points. It is also used for determining the elevation value of a terrain point being affected by a river.

By selecting any of the river-points in the hierarchy-window, will display information relevant to that exact point, demonstrated in Figure D.6. It is possible to override the default radius and depth of the river, for a river-point by simply enabling those check-boxes and inputting a new value. Clicking the "Add point"-button on a river-point will extend the river by attaching a new river-point, to the one currently selected.

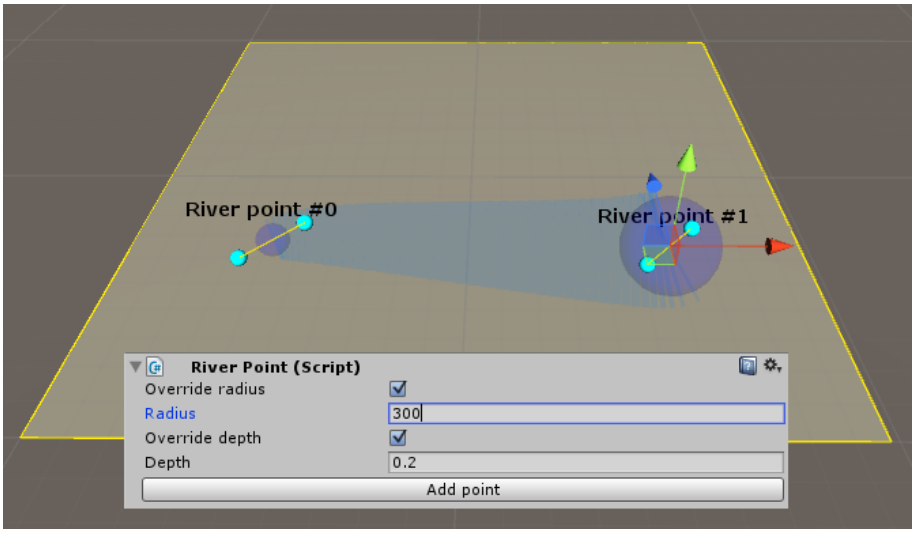


Figure D.6: A river-point is selected and its default values overwritten.

The various river-points which make up the river can be moved around by simply clicking and dragging their gizmos (blue spheres) around. Likewise, every river-point has two handles connected to them, which can be freely manipulated by simply clicking and dragging them (teal spheres) around the scene. The positions of the river-points and the positions of their handles determine the path of the river.