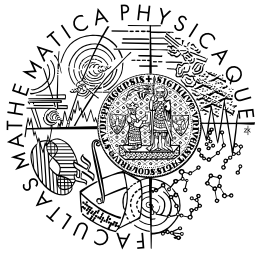


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jaroslav Knotek

Procedural terrain modeling using polygonal sketching

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Martin Kahoun
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

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

In ..... date ..... signature of the author

Title: Procedural terrain modeling using polygonal sketching

Author: Jaroslav Knotek

Department: Department of Software and Computer Science Education

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

Abstract: This work presents a method of virtual terrain sketching using polygons where each polygon represents a feature of the terrain. Each feature is generated to a layer and the resulting terrain is then obtained by merging those layers. It can be further edited by convolution brushes, eroded, or have a river drainage generated. The final result can be previewed in 3D. The implementation allows a user to save and load the project as well as export the final height map.

Keywords: procedural modeling, virtual terrain, erosion simulation, river networks

I would like to thank my supervisor Mgr. Martin Kahoun for his patience and guidance.

Contents

Introduction

The quality of 3D graphics has increased significantly over the past decades. Its applications range from visualizations of architecture yet to be built, over medical imaging to entertainment where computer generated imagery is used for special effects in movies as well as for game world representations. But creating content by hand can consume a lot of time and money. Let us take an example of GTA V that features a virtual world of a size comparable to San Francisco [1]. Its development cost is estimated to be around 143 million dollars adjusted to 2017 inflation, which makes it the second most expensive game ever developed [2]. It is possible to create a world of the same size with much lower price and effort using procedurally generated content — that is a method of creating data using algorithms.

Content made by hand requires skilled artists who can use specialized software, money and time to meet strict quality demands. The larger the created world, the more artists, software and money is needed to meet the same quality standards and deadlines. On the other hand, using procedural generation requires only initial development costs, which is a crucial factor for the present game industry and its independent scene.

There are many tools that can generate content such as terrains, buildings, cities or even sounds that are very hard to use. A user usually needs to spend much time reading documentation, watch online tutorials or try to achieve any result by the trial-and-error approach, which is an unacceptable drawback of these tools when time is an important factor. This is for example the case of Terragen[3] — a software that allows users to develop detailed and photo–realistic sceneries when used skillfully. But the skillfulness is too time consuming to gain.

Therefore, the goal of this work is to create a free tool that makes procedural terrain generation simple and easy using the sketch-based approach. The program is called TerraSketch and it conceptually derives from an approach presented by Smelik et al. [2010b]. However, due to the scope of this work, TerraSketch is offered with very limited features and it focuses mainly on simplicity.

In the application, a user interacts with three different screens also referred to as views. In the first view the user can define a polygonal area or areas representing the terrain features such as a flat area, a canyon or a mountain and set parameters that alter their appearance. These are namely the level of detail, the order in which the field is rendered and erosion. When the terrain is generated, the user can slightly edit the heightmap using convolution brushes such as blur, sharpening, elevation and lowering. The last view serves as a preview of the terrain's 3D model. The final product can be exported to a machine readable format and to make development easier, TerraSketch allows the user to save or load the project at any time throughout the development process.

This work is structured as follows. Chapter 1 offers a brief overview of procedural generation in general and then focuses on terrain-specific approaches. The problem analysis and details of the chosen algorithms are discussed in Chapter 2

[1] `http://tinyurl.com/gta-v-los-santos-size`

[2] `http://tinyurl.com/gta-5-development-costs`

[3] `http://planetside.co.uk/terragen-overview/`

giving the reader a basic idea of the process that begins with a random noise function, through the erosion simulation, resulting in the final terrain object. Chapter 3 describes the flow of the application in detail. The user is introduced to the process of terrain features definition, layer generation, heightmap adjustments and 3D previewing. The results and future work are discussed in Chapter 4. Appendices cover the user guide in Appendix A, code documentation in Appendix B and save file structure in Appendix C.

1. Procedural modeling

Procedural generation is a process that uses a set of input parameters and a given algorithm to create any type of content such as sounds, textures or 3D objects. Procedurally generated content can make the development easier and reduce its costs at the same time. Many games feature vast terrains that can be difficult to create. When procedural generation is employed, the content can be produced in a moment with the use of a proper tool. This can be developed either in-house (which increases the costs) or an existing tool may be applied. When the generator is prepared and the product generated, artists can spend time only on a potential refinement of details or on adding elements that were not generated. Procedural generation has advanced significantly, but many games still use procedural generation to obtain only a basic product that is further elaborated.

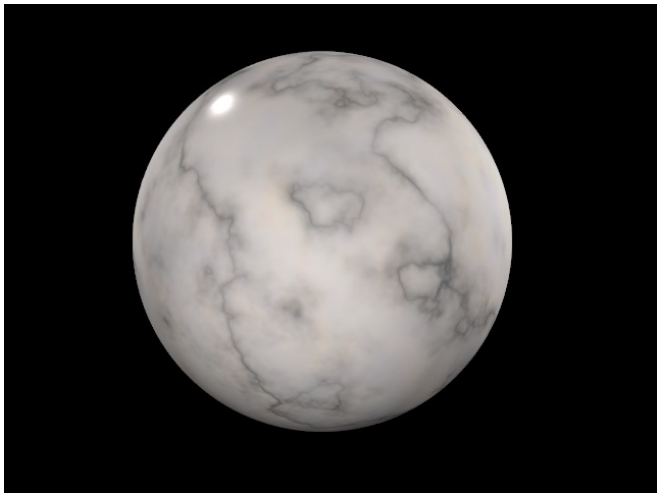
Procedurally generated content ranges from simple textures (as shown in Figure 1.1a), through more complicated objects such as plants (as can be seen in Figure 1.1b) produced by L-System described in a book by Prusinkiewicz [1996], to whole cities (see Figure 1.1c) as an example of a complex human made object. Procedural generation is not restricted only to static objects. For example, Gritt [2010] explores an artificial movement of fish and generates its animations. Many other procedural methods exist, for an in-depth overview of the topic we refer the reader to a recent survey by Smelik et al. [2014]. In the following text, we will focus on terrain generation.

1.1 Terrain generation

Terrain generation can be based on persistent noise functions such as Perlin noise Perlin [2003], which, in its basic form, is not very useful due to its smoothness. It does not look like a terrain, but it can be used to create a more advanced and more controllable noise used to generate the basic terrain that is later altered through transformation. To create a more interesting terrain we can warp the noise function and subject it to various transformations or apply more advanced techniques such as erosion simulation and river drainage carving. For more details on procedural fractal terrains see Musgrave [2003a] and on building planetary bodies see Musgrave [2003b].

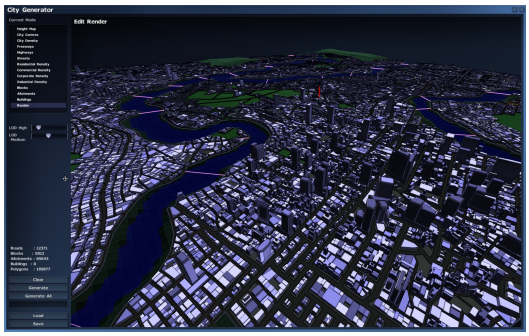
The generation can become very complex when various biomes and terrain features are present in one terrain created by a single function such as those presented in the work of Kahoun [2010] that focuses on a planetary generator. This method does not allow users to generate one part differently from the other, which means that it has to be edited afterwards.

To give a user more control over the generation process, we can adopt a more imperative approach. The user can specify the behavior of a generator for given features by parameters that the generator uses during their generation. Other features without parameters are generated by a general process. In some cases, the user may want to focus on the definition of very specific features that the random generator would not generate itself. For example, the user may want to create a terrain for a multiplayer strategy game where the player's starting position should



(a)

(b)



(c)

Figure 1.1: Examples of procedurally generated content: a) a marble texture, b) a plant, c) a city. Sources, from left to right, top to bottom: `https://commons.wikimedia.org/wiki/File:Blender3D_MarbleExample1.jpg`, Algorithmic beauty of plants by Prusinkiewicz [1996], `http://www.introversion.co.uk/blog/20100221/screenshot1.jpg`

be similar to the starting position of other players. A drawback of this approach is that we need to maintain a balance between control and consistency because the user-specified part has to fit or blend into generated ones.

Constructive solid geometry (CSG) can be used as an intuitive way of blending terrain primitive. CSG is a geometry built up with basic boolean operators and terrain primitives. It is represented by binary tree where primitives are the leaves and internal nodes represent operations on them. This topic is described in detail in the book Computer Graphics: Principles and Practice by Foley [1996]. The idea is adopted in a work by Zábský [2011] who created a scriptable terrain generator that uses simple terrain primitives and combines them to obtain a complex result.

CSG allows users to create a very detailed terrain using a combination of various connected terrain primitives (also referred to as components). This approach results in a more complicated definition process and much more complex graphical user interface. To obtain a result, the user has to spend a lot of time defining and connecting components together. This approach is implemented in various procedural generators such as Terragen, Houdini[1] or World machine (discussed in Section 1.2) that are well-known for their complexity.

In order to create terrain with a real-world appearance and feeling, the effect of thermal and hydraulic erosion should be simulated. The terrain generated using noise methods looks like a terrain from a paleozoic era for its sharp peaks and non-eroded terrain. Thermal erosion is a process in which a material falls apart due to thermal effect, which basically means that the material crumbles until it is a pile. Hydraulic erosion is a process in which water transfers sediment to other parts of a terrain depending on the water flow Cojan and Renard [2002].

One more feature that can contribute to believability of generated terrains is water drainage. The river path can be found, for example, by tracing steepest descent from a certain point (river spring) to the terrain edge or a local terrain minima, thus creating a lake. If no suitable path is found, the algorithm can alter the terrain to suit the river flow. An alternative approach is to create a river network first and then generate the terrain around river basin. This bottom up approach usually has more natural result than finding rivers on an existing terrain. The algorithm is explored in detail in the work of Belhadj and Audibert [2005].

Another interesting bottom-up approach is described in a paper of Genevaux at al. Génevaux et al. [2013]. A river network is generated using L-system from river mouth to its spring. Then a map is divided to Voronoi cell that represents river primitives that determines basic river flow. The map is then generated using noise functions and a CSG tree that combines river primitives and terrain primitives. The whole terrain is represented strictly by the CSG tree.

Terrain size varies from the smallest like used in architectural project that needs terrain for immediate surroundings through a large-scale terrain that is used in games, to an infinite terrain generated runtime used in a game Minecraft or described in the work of Parberry [2014]. Some games in need of infinite worlds can resort to implementing tileable terrains. Tileable terrain is a terrain split into square tiles where adjacent tile edges have the same elevation in order to create seamless transition between tiles.

[1] `http://www.paulwinex.ru/vex_editor_1_0_5/`

We will now discuss two of the most common terrain representations used in procedural generation. The first one is based on a two dimensional array of values that represents height on the specific coordinate called heightmap. This representation does not allow to store data about caves, cliffs or any other feature that is above or beneath ground but it is easier to work with.

The other representation is voxel-based which is means that data is stored in three dimensions instead of only two. Each voxel of the three dimensional space contains information whether the point is solid or not. Shamus Young in his work Young [2012] builds his own voxel based framework from scratch and explains each step. Implementation goes bit deeper than just 3D array of bits. Deterministic noise function is used to generate infinite 3D space but generates and shows only visible cells. In a post-process slope simulation algorithms are employed to ensure that the final product is not edgy but smooth which has only visual impact. A result of this approach is shown in the Figure 1.2.

TerraSketch stores terrain data in a heightmap, because it is simple and easy to work with. Also overhangs can be added after a heightmap is generated to its model as static meshes, or the terrain mesh can be edited directly.

Advanced methods use a cooperation between generated layer such as, a terrain influences road network which influences building placement (See Smelik et al. [2011]). Some generators allow road to cooperate with a terrain to produce more realistic results. Unreleased prototype by Introversion Software called Subversion (as seen in the Figure 1.1c) generates a large city using hierarchy of nodes and recursion. Each type of content is a node with its own parameters. The first step is to generate root which is a terrain with lakes and rivers. It's children are cities centers and suburbs that has to be connected using road network. With city layout done, generator assigns each city part its type such as commercial, corporate, industrial or residential which has its. Each area has different building types and styles. Unfortunately, this project was closed 2011, however, a source code is available for sale.

1.2 Related work

Several procedural terrain generators already exist, furthermore some of them use a sketch based approach as an intuitive way to define parameters for a patch of a terrain defined by a polygon or curves. For example SketchaWorld (as seen in the Figure 1.3) which is a software, that allows user to add characteristics to a terrain using a polygon or a regular grid where parameters are added to each cell separately. World Machine (as shown in the Figure 1.4) also uses a sketching to support more detailed characterization but it is optional, user can generate terrain entirely without using sketching, but the result will look artificial and will lack the level of detail seen in real world terrain.

World machine

A work-flow of a TerraSketch resembles to a work-flow of a software World Machine which also features separate views for the polygonal definition (as seen in the Figure 1.4) of a terrain primitive which has various parameters. Some primitives can be edited with brushes as well. The terrain can be previewed in a

simplified version in a real-time and a final version must be rendered with much more effort.

The terrain is defined through nodes called components which have input and output slots where output of one component can be connected to an input of some other component. Components together form a graph where various simple components creates a complex structure that give user a perfect control over a process of generation. Components might be a noise function, a terrain primitive with a certain influence such as elevation, river basin model or it could be a component that simulates an erosion.

World Machine began as an experimental application that resembles to crude terrain generation software and is still under active development by Stephen Schmitt. It is available under commercial licence but it can be used free of charge for non-commercial purposes.

SketchaWorld

This project focuses more on the creation of whole virtual worlds — terrain is just one of several layers composing the final landscape. Apart from terrain a user can define roads, rivers, forests and urban areas. In SketchaWorld, each aspect of a world cooperates with other as much as possible in order to preserve realism. For example: a user sketches a river using control points and algorithm tries to find a way through all the given points without changing the terrain but in some cases it is needed to alter some features. For roads that cross the river bridges are automatically created. A drawback is that this approach does not allow users to create details since they can only define macro properties.

The most interesting feature is that virtual world can be constructed from an aerial photography. Interaction with a user is required to properly define unrecognized object on photography.

This project was supported by the GATE project, funded by the Netherlands Organization for Scientific Research (NWO) and the Netherlands ICT Research and Innovation Authority (ICT Regie) and was developed in collaboration with researches from TNO Smelik et al. [2010a]. Last information about this project comes from Smelik et al. [2010b].

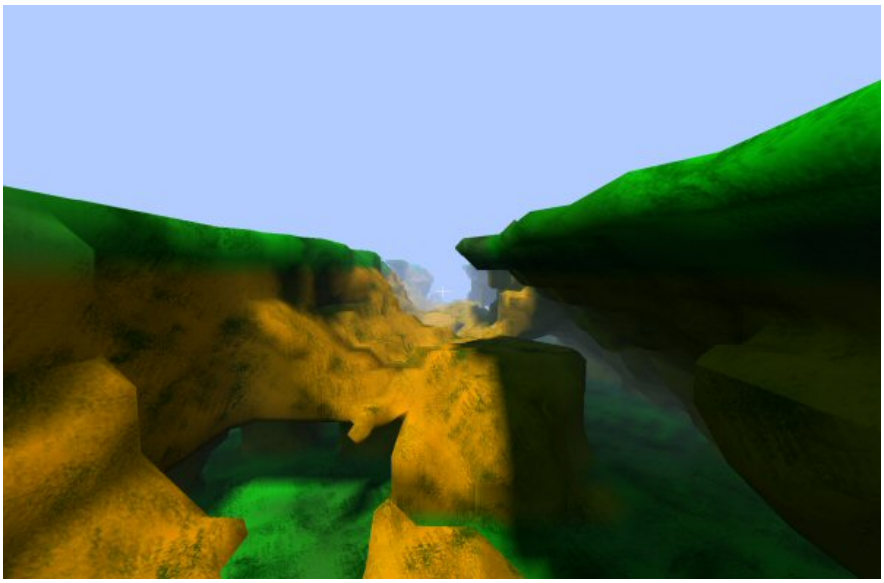


Figure 1.2: Voxel terrain from project Octant by Shamus Young `http://www.shamusyoung.com/twentysidedtale/images/preview_octant.jpg`

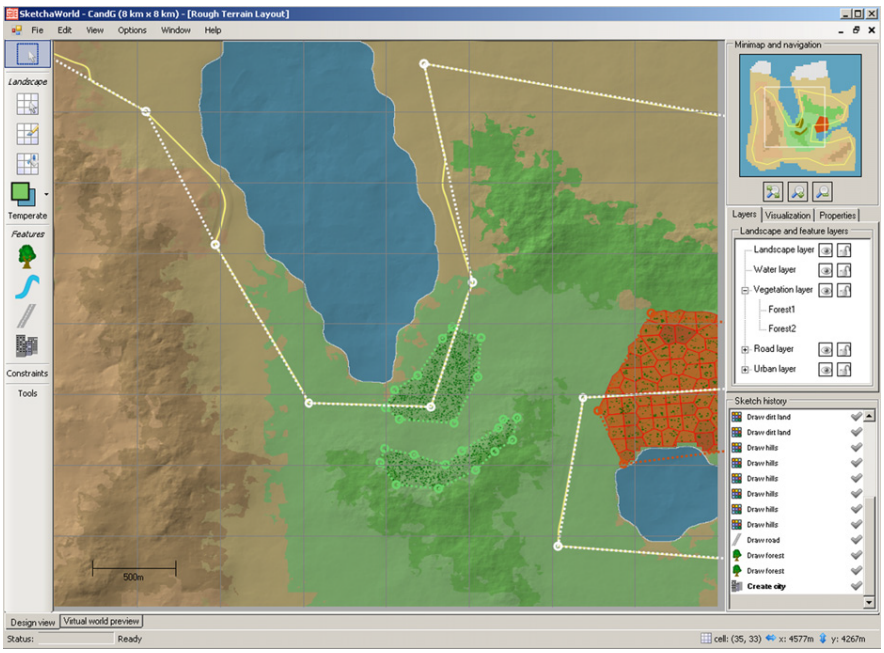


Figure 1.3: User interface of SketchaWorld. Source: `http://www.world-machine.com/about.php?page=features`

Figure 1.4: A polygon definition window in World Machine. Source: Smelik et al. [2010b]

2. Problem analysis

TerraSketch is a program that performs texture synthesis to obtain a terrain heightmap which is consequently processed and visualized in 3D. For that reason, it employs basic algorithms and methods used in image processing such as convolution, blur or interpolation and the generating part employs basic noise generation function that is in some cases enhanced with rendered Voronoi diagram Aurenhammer [2013] which in combination makes an impression of a terrain when rendered in 3D. After an erosion simulation is performed and river drainage added, the primitive obtains a real world terrain appearance.

2.1 Noise

The terrain generation is usually based on a persistent noise function that looks seemingly random but queries for the same coordinate result in the same value. In our case Perlin noise Perlin [1985] is employed for its simplicity and controllability. Perlin noise function was developed by Ken Perlin who described the algorithm in detail in his book Texturing and Modeling Perlin [2003]. It works for any dimension but in the following text we will explain Perlin noise for two dimensions only.

The first step of the Perlin noise algorithm is to generate a two-dimensional grid of random gradient unit vectors G. To negate expensive random vectors computation, an array of vectors can be precomputed. Now, for a given point P we determine a cell C of G to which P falls.

For P we determine distance vectors d_1, d_2, d_3, d_4 to each corner of C. For each gradient vector g_1, g_2, g_3, g_4 of C we compute a value v_i from a dot product of vectors g_i and d_i:

$$v_i = g_i \cdot d_i.$$

The final step is to interpolate between values v_i using a two-dimensional cosine interpolation.

Perlin noise itself is not suitable for terrain generation because it creates bumps that are unnaturally smooth. To create a more interesting texture, fractional Brownian motion is employed.

Fractional Brownian motion (abbreviated as fBm) is a process of computing a value by combining values of a general persistent noise with different amplitude a and frequency f. The following equation computes a value for a point x, y, noise $g(x, y)$, amplitude a_i of a frequency f_i, function $h(x, y)$ returning height of the point x, y and n, which is a number of sampled frequencies:

$$h(x, y) = \sum_{i=0}^{n-1} a_i \cdot g(f_i \cdot x, f_i \cdot y)$$

The frequencies f_i and amplitudes a_i may vary significantly, but usually amplitudes a_i and frequencies f_i are given by equations $a_{i+1} = \frac{1}{2}a_i$ and $f_{i+1} = lf_i$ where l is called lacunarity (Latin word for gap) which, together with amplitudes, controls the roughness of the resulting noise. Lacunarity controls a difference between sampled frequencies and amplitude controls addition of each frequency to the final noise. The best results are achieved when $l = 2$ and $a = 1/2$.

FBm with recommended parameters creates an interesting texture that resembles the terrain of the Paleozoic era when rendered. To create a look of the present-day terrain, we begin with adding details to some parts of the terrain to create mountains using a Voronoi diagram with distance transformation.

2.2 Voronoi diagram

The noise lacks any particular shape, but combining it with a transformed Voronoi diagram that adds distinctive ridges and valley, we get a terrain looking like mountains. A Voronoi diagram creates a set of convex hulls for a set of distinct points $P_1 \dots P_n$ distributed on a plane P. All points that are closer to a point P_i than any other are called Voronoi cell $V_c(P_i)$.

$V_c(s) = \{x : dist(x, s) \leq min_{z \in \{P_1 \dots P_n\}} (dist(x, z))\}$ where $dist(a, b)$ is a given metrics, our implementation uses Euclidean metrics. Voronoi diagram for a set $X = \{P_1 \dots P_n\}$ is $V(X) = \{x : V_c(x)\}$.

One possible algorithm that computes a Voronoi diagram is a Fortune algorithm running in $\mathcal{O}(n \log n)$ which was originally published by Steven Fortune Fortune [1986]. It is classified as a sweep line algorithm, which means that it gets a set of points $P_1 \dots P_n$ and sweeps the plane meaning moving a line L from top to bottom and performs an action on each point.

Supposing that points $P_1 \dots P_n$ are sorted lexicographically by (x, y) coordinates, sweeping may begin. Each time L hits a point P_i a new parabola p is created with focus in a P_i and L as a directrix. This situation is called site event shown in Figure 2.1a.

The next step is to determine an intersection x with a beach line B (drawn thick black in the Figure 2.1a) that consist of parabolas $p_0 \dots p_k$ created during sweeping and intersections $x_0 \dots x_l$ of two consecutive parabolae p_i and p_{i+1}. Beach line can be expressed as $B = (p_0, x_0, p_1, x_1 \dots x_k, p_{k+1})$ where intersection x_i has the same distance to foci of parabolae that creates it: $dist(x_i, f(p_i)) = dist(x_i, f(p_{i+1}))$ where $f(p)$ returns focus of the parabola.

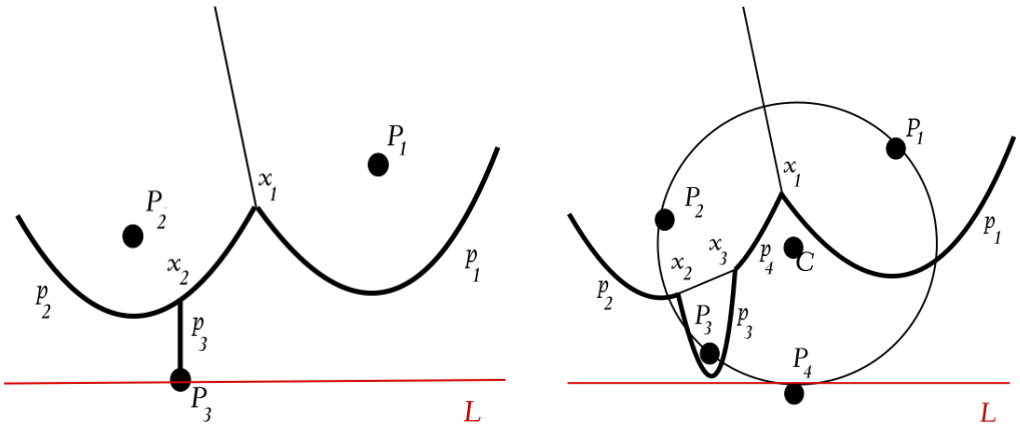


Figure 2.1: Diagrams of events: a) a site event, b) circle event — which creates point P_4. P_i are points, p_i parabolae, x_i intersections of parabolae. The beach line is shown as a thick black curve.

When the intersection $x = p_i \cap p_{i+1}$ is found we add p to the beach line so $B = (p_0, x_0 \dots p_i, x, p, x, p_{i+1} \dots x_k, p_{k+1})$.

Beach line is also altered when p_i and p_{i+2} "consume" p_{i+1} which is called circle event shown in Figure 2.1b. The parabola p_{i+1} disappears along with x_i and x_{i+1} thus new intersection $y = p_i \cap p_{i+2}$ appears and $B = (p_0, x_0 \dots p_i, y, p_{i+2} \dots x_k, p_{k+1})$. Disappearance of any parabola encloses a Voronoi cell. The implementation is explained in detail in the book Computational Geometry Berg [2000].

A Voronoi diagram can be rendered to a texture image. The renderer assigns to each pixel a value according to a Euclidean distance to the nearest edge of a cell. The result of this diagram can be seen in Figure 2.2.

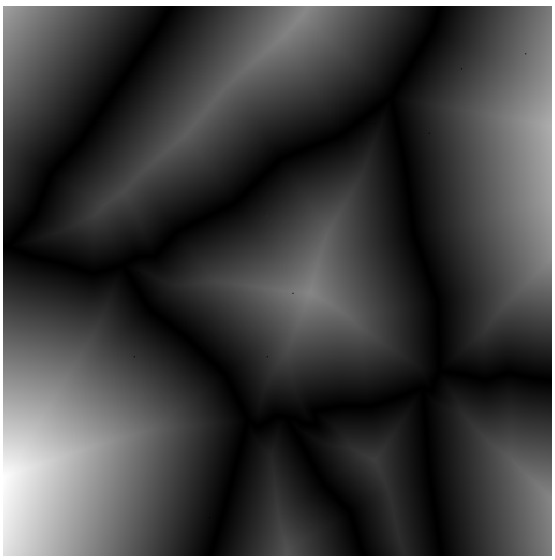


Figure 2.2: Mountain profile

A terrain generated using fBm and Voronoi diagram does not have an appearance of a present-day terrain which has been exposed to long-term erosion. There are two types of erosions: hydraulic and thermal. Hydraulic erosion is a process in which rain falls, takes sediment and transports it downhill while thermal erosion breaks down material by thermal effects. TerraSketch implements hydraulic erosion only.

2.3 Hydraulic erosion and river simulation

The implementation is based on the work of Olsen [2004] who offers a solution simulating the long-time effect in a real world with various optimizations that significantly improve its performance.

The hydraulic erosion algorithm works with following two-dimensional arrays terrain T that is a given heightmap, water map W which stores temporary water levels and a sediment map S that stores transported sediment. We will also need constants for dirt solubility C_s multiplying water column resulting in the amount of sediment to be transported, rainfall per iteration C_r and evaporation constant C_e which affects a portion a water that evaporates each iteration and leaves sediment behind. The lower C_s the more iterations are required to affect

the terrain, which significantly slows down the algorithm, however, a great value results in an unrealistic result.

The algorithm runs in the following steps. The first step of the iteration fills the water map W with rainfall C_r. If a river drainage map is generated, it can fill W in each iteration to naturally enhance river basins.

In the next step water absorbs an amount of sediment D that depends on a C_s:

$$D = min(W(x,y), T(x,y) * C_s).$$

Now we have to determine the lowest neighbor. This step is simplified since water would flow to all lower neighbors but in our case we use only the lowest one to optimize performance. Once the lowest neighbor is determined one of the following situations occurs:

1. The height $h_1 > h_2 + w_2$, which means that water is transported completely as seen in Figure 2.3a. In this case water on coordinates x,y is determined by the following equation:

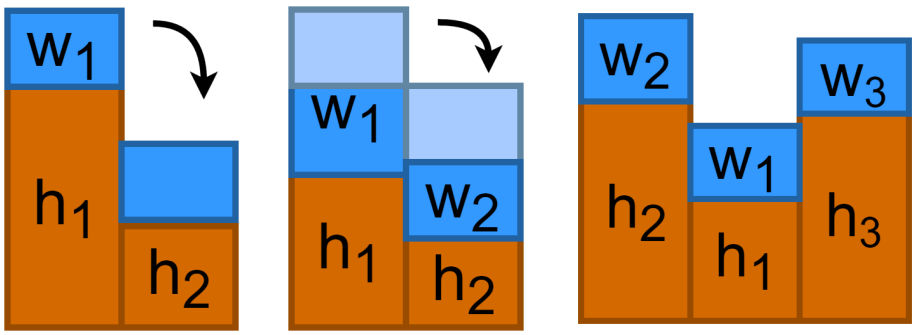
$$W(x,y) = W(x,y) + w_1.$$

2. Water w_1 is transferred partially to the neighbor so the water level is evened out by adding one half of the difference to the lower water level w_2 as seen in Figure 2.3b.

$$W(x,y) = W(x,y) + ((w_1 + c_1) - (w_2 + c_2))/2 \tag{2.1}$$

3. The height of a cell c with its water w is less than the heights of neighbors so no water is transported at all. The situation is depicted in Figure 2.3c.

4. A certain amount of water W evaporates leaving some sediment to S according to C_e constant.



(a) (b) (c)

Figure 2.3: Cases of water transportation: a) a complete transport of water, b) a partial transport of water, c) no water transported.

The terrain on which an erosion simulation was applied looks more realistic. To continue reaching the realistic look, we will create rivers that eventually create

larger water bodies along the way of their flows. The details of their implementation are discussed in Section 3.4.

The idea of the algorithm comes from the work of Belhadj and Audibert [2005] who generates a river network on a map influenced by heights of ridge lines only. The mobile river particles r_i with acceleration and mass, which is subject to gravity acceleration, are used to simulate the river flow along with its depth and width. Moving river particles create a river basin, when r_i intersects a basin created by r_j, r_i is stopped and r_j is backtracked to the intersection points changing its velocity

$$velocity(r_j) = velocity(r_j) + velocity(r_i)$$

and mass

$$mass(r_j) = mass(r_j) + mass(r_i).$$

Generated river paths are stored to a drainage map along with a height in each coordinate. When the river drainage map is ready, the terrain detail is enriched with midpoint displacement algorithm (described in the book Modeling and Texturing Musgrave [2003a]). Our implementation is a simplified version of this algorithm that does not support mass and acceleration.

The algorithm starts with a given river source s that we put to a heap H that stores information about the lowest processed heightmap coordinates. Starting with taking the river source from H, we take all its neighbors in Moore neighborhood $n_1, \ldots, n_8$ and add them to H. Then we continue taking the coordinate of the lowest point l from H and adding all its neighbors until the current processed coordinate is on an edge of the map. This process simulates water behavior in which water flows in a direction with the steepest fall and when water stops flowing, it slowly starts to fill the hole in which it is stuck until it finds its way out.

An initial implementation of the river simulation was based on an algorithm proposed by Zelený [2009]. The algorithm creates a river drainage map by calculating what he refers to as primary rivers. Finding primary rivers of a terrain H begins with adding a small constant c to each point h of H, then for each h we find all neighbors $l_1 \ldots l_i$ for which h is the lowest neighbor, sum their values and set them to h

$$h = c + \sum l_i.$$

After many iterations this produces a map of drainage area sizes. Rivers can be extracted by any edge detector such as Canny edge detector Canny [1986] or a Sobel operator Sobel [2014] and then reconnected, which creates primary rivers. The secondary rivers and lakes are generated selecting a river in the lower attitude, iteratively extend its size so it becomes a lake. If the lake overflows, it creates a secondary river that can create another lake.

This river simulation method does not rely on geological accuracy at all, which is the main reason why we implemented a different algorithm. The initial evaluation of this method proved to be insufficient due to the fact that rivers do not look natural enough. For example intersections create really artificially looking confluences shown in Figure 2.4.

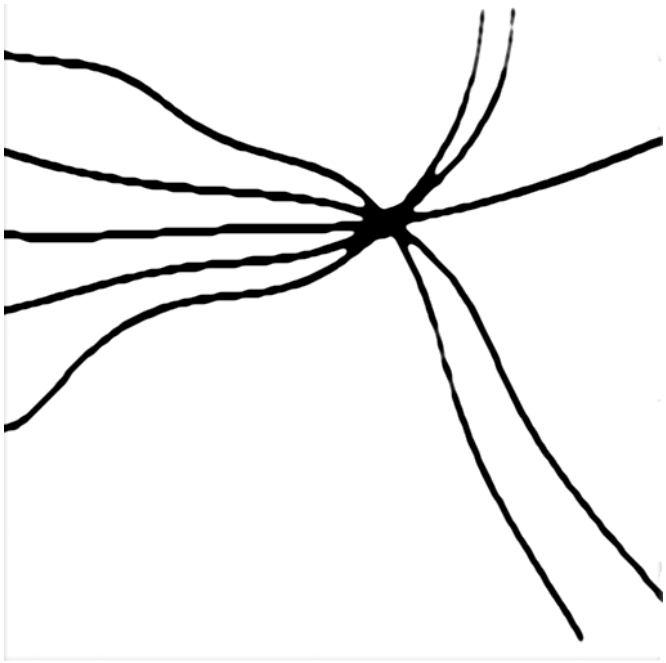


Figure 2.4: Unnatural river confluences produced by our implementation of a method by Zelený [2009]

3. Implementation

TerraSketch allows a user to quickly and easily create a terrain using polygonal sketching. Each polygon represents a layer that has several parameters affecting the generation process. The generated terrain can further be enhanced with convolution brushes and the result immediately previewed.

The program features three main tabbed views. The first view lets users to define a polygon and parameters, the second allows them to change the generated terrain with brushes and the last shows the result in 3D. This layout allows users to focus primarily on the specific scope of a work. The user does not have to rearrange windows or controls, just switch tabs.

As discussed before, the final terrain is composed of several blended layers. Before a layer is produced one must define its shape, position, and customize its parameters. This concept allows users to focus precisely on specific parts of a terrain and spend less time on all other parts by setting global parameters of a base layer.

The shape is given by a polygon that does not intersect itself. Its points are defined by clicking on a canvas of the first view. Polygon points can be selected separately, which helps us select specifically the points that we want to translate, rotate, scale, subdivide or merge.

3.1 Terrain layers

The user can alter the appearance of a patch of the final terrain with layer parameters such as profile, level of detail, blend mode, offset and seed. Seed is a parameter used in a random generator initialization completely changing its output. Seeding is a useful method for obtaining the same result for the same seed if parameters do not change.

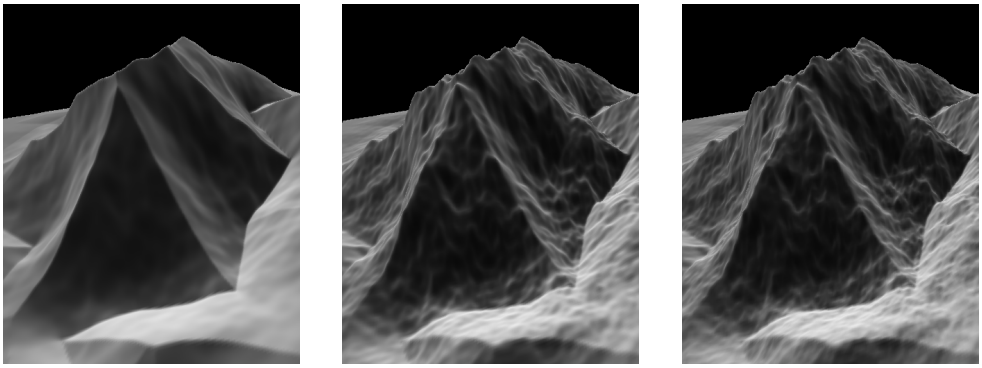
The vertical position of a layer is affected by the offset parameter allowing the user to create a plateu by setting a high offset to a flat terrain, for example, or to adjust height of a mountain. The offset is a value ranging from -1 to 1 that is added to heights of a layer. Its effect is limited but it was designed only for minor adjustments. Greater values can result in unnatural results

The blend mode is a method of blending two layers together. It works with two layers: F, G where F is a layer into which G is blended. TerraSketch offers following blend modes: *replace*, *minimum*, *maximum*, *add*, *subtract* and *invisible* (does not render the layer, which is useful for visualizing the difference between a map with the layer and without it). Each blend mode has its mathematical representation (see Table 3.1), since a layer can be considered as an approximation of the 2D function $F = f(x, y)$, $G = g(x, y)$.

Level of detail changes how much detailed the result will be in terms of 2D generation, it does not alter the 3D model. The terrain can be very large and can have many features only visible from a long distance. In this case, there is no need to spend much time on the generation process and we can reduce the output quality by lowering the *level of detail* parameter. In implementation it simply means how much effort the algorithm should spend on a result, which in our case means how many octaves a noise generator will have. Various *level of*

Blend mode name	result given by
Add	$f(x,y) + g(x,y)$
Subtract	$f(x,y) - g(x,y)$
Minimum	$min(f(x,y), g(x,y))$
Maximum	$max(f(x,y), g(x,y))$
Replace	$f(x,y)$
Invisible	$g(x,y)$

Table 3.1: Blend modes



(a) (b) (c)

Figure 3.1: Level of detail settings: a) low settings, b) medium settings, c) high setting

detail settings can be observed in Figure 3.1.

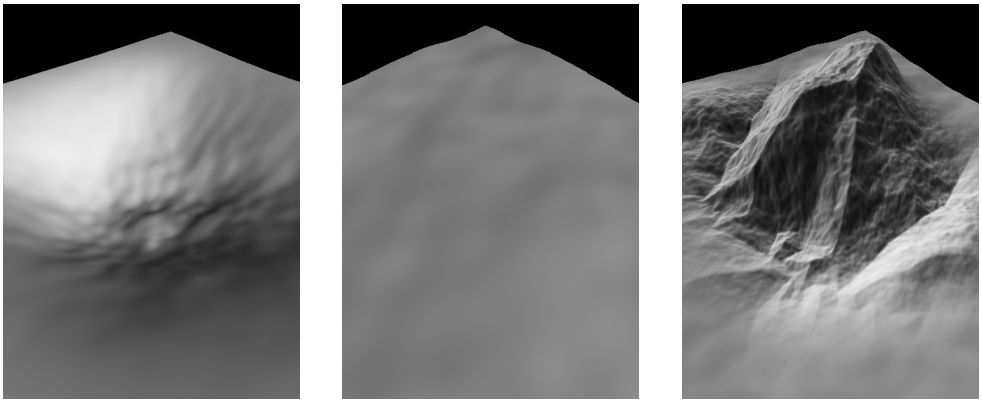
Profile is a parameter that has the most significant impact on the layer, because it implies what kind of generator will be used. Each generator can use a completely different process to generate the layer. The following text explains the structure of a generator, its process of terrain generation and its implementation.

3.2 Layer generators

In order to facilitate the implementation, a more complex algorithm consisting of several noises and transforming functions can be encapsulated in a generator object. The generator can contain other generators as an input allowing to reach a complex result with less effort. The generator is designed to be customizable by user, but the current version does not support it. In the current implementation, each profile has a generator assigned that is created during the build.

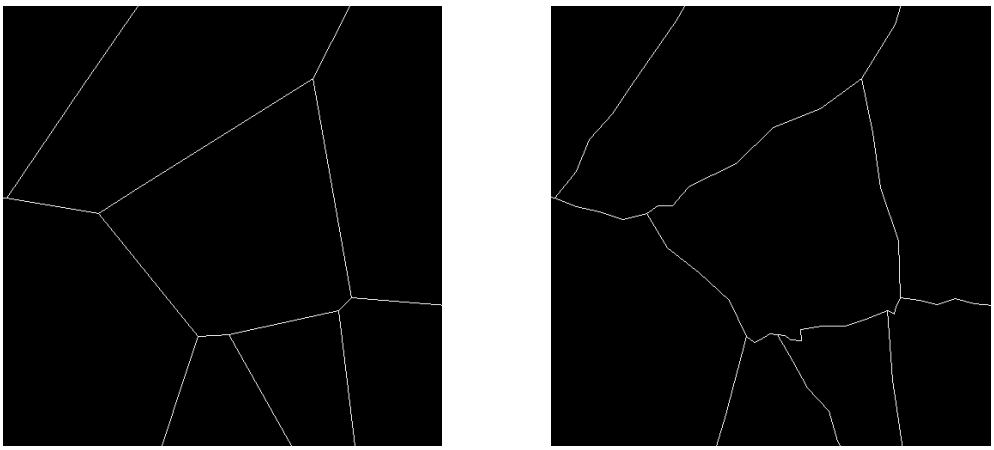
A mountain generator obtains a basic mountain profile employing a Voronoi diagram with distance transformation. The Voronoi diagram described in Section 2.2 generates only straight edges as seen in the Figure 3.3a. This appearance is unsatisfactory since it creates unnaturally looking mountain ridges.

To prevent those artifacts, we subdivide each edge e into several segments with endpoints p_i about 150 pixels long, then we get a vector n perpendicular to e and normalize it. The vector n is then multiplied by a random number a ranging from -25 to 25 and added to the endpoint of each segment resulting in a new segment (see Figure 3.3b) with endpoints r_i given by the formula $r_i = p_i + an$.



(a) (b) (c)

Figure 3.2: Example of terrain profiles: a) canyon, b) flat, c) mountain



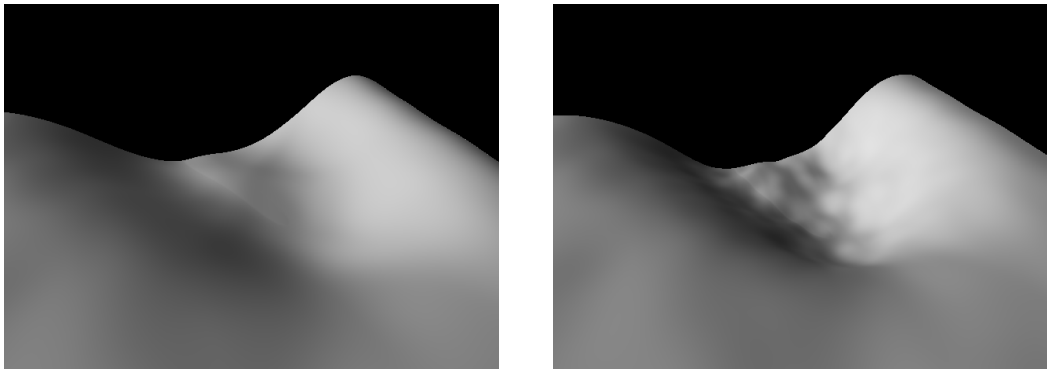
(a) (b)

Figure 3.3: Voronoi diagram edges enhancement: a) without, b) with subdivision and jittering.

The subdivided edge is then used as a ridge line to generate a less artificial shape which gives a rough impression of mountains. To make it more naturally looking we add a noise by generating it with 6 to 12 octaves, depending on a user parameter selection, in a new layer. The layers are blended together while the noise will add a small-scale detail and cover up the sharp edges.

A flat terrain is a fractal noise only scaled down to one fifth, which gives the impression of a flat terrain that is not artificial. It has proven difficult to create a flat terrain that is customizable by level of detail and still looks natural. The detail settings affect mainly the amount of octaves, which has impact on the bumpiness of the terrain. A flat terrain is not supposed to be rough, meaning that in all settings only a low frequencies with small amplitude are sampled which does not allow to create very noisy image. The result of this approach can be seen in Figure 3.2b that has a visible noise pattern that still looks flat enough, like very low rolling hills or a wavy landscape.

Canyon profile has a process similar to the mountain profile generator. The first step is to create a gradient that creates a large hole using a distance-transformed Voronoi diagram with the only difference being the fact that the



(a) (b)

Figure 3.4: Different stages of canyon generation: a) a canyon gradient layer, b) a final canyon.

canyon profile uses only one Voronoi cell the shape of which is given by the input polygon. The result of this approach is visible in Figure 3.4a. The picture also shows a small artifact caused by the masking process. The artifact is visible as sharp edges at the bottom of the canyon and is caused by blurring process. In order to cancel out the artifact and add some detail a noise is generated and added to the gradient to create a less artificially looking canyon shown in Figure 3.4b.

3.3 The generation process

Some parameters are used in the generation and other during merging. The parameter profile implies a generator that will be used. The generator is initialized with *level of detail* and *seed* that are converted to a code affecting a generation process. Initialized generator is stored along with *offset* and *blend mode* in the field. See Appendix A for more implementation details.

The translation of a selected *level of detail* to noise parameters was in a previous version realized by mapping some *level of detail* to global values. This solution proved to be insufficient since high detail setting should have different impact on a mountain than on a flat terrain. The current solution for mountain is shown in Table 3.2, a flat terrain and a canyon share settings shown in Table 3.3.

Level	Number of octaves	Amplitude	Frequency
Low	6	0.4	2
Medium	7	0.45	2
High	8	0.5	2

Table 3.2: Mountain detail mappings

With parameters translated and the generator set we can approach to an actual generation, but before it starts, each layer must have its polygon slightly upscaled. Enlarged polygon is used to define transition area for a mask. The area inside the original polygon is visible, everything outside the enlarged polygon is invisible and the area between serves as a transition.

Level	Number of octaves	Amplitude	Frequency
Low	3	0.4	2
Medium	4	0.45	2
High	5	0.5	2

Table 3.3: Flat and canyon profile terrain detail mappings

Mask is another layer, which values range from 0 to 1, applied as a filter for another layer saying how much is any pixel visible where 0 means pixel of the terrain layer is invisible and 1 means that it is visible and it completely hide the pixel under.

Mask is created by a rasterization of a polygon that is further blurred to smoothen sharp edges. The first step of the rasterization is to draw all polygon lines using a Bresenham's line algorithm Bresenham [1965]. Then we find any point outside, which is very simple, and then use a flood fill algorithm to mark all pixels outside of the polygon. The layer on which the polygon is drawn is slightly larger to prevent incorrect polygon detection.

The mask with rasterized polygon is still not suitable for its sharp edges. For this reason, we have to blur them. Each polygon edge was moved by a constant distance e ffrom the center in the edge normal direction. Blur with $c \times c$ kernel is used to create smoothen those edges. The values e and c are implemented as constants in the current version. Making them variable would be simple, but we would have to check that $e > c$. Otherwise the mask will not fade out soon enough resulting in a sharp edges on layer borders anyway.

Masking is a useful method for a seamless layer blending with no significant artifacts but our implementation is not perfect. In some cases there is a problem in blur algorithm that causes sharp edges to appear in the middle of the layer. This problem can be solved with multiple blur application.

The positives of masking is that it can be customized in each pixel which means that the transition between two layers might be other than interpolation function. It could be another two-dimensional array with the same size filled with values ranging from 0 to 1. So it could be customized by user to achieve even more precise results.

When the mask is ready, the terrain layer is generated using its own preset generator described previously in Section 3.2. All layers are generated separately. The generation process of a generator may vary significantly depending on a selected profile but in general, all of them share a common flow shown in Figure 3.5. Generators in general generate a main layer, but they can possibly contain other generators that blend their own result to the main layer. The whole process results in an ordered set of masked layers.

Layer contains values in a range that depends utterly on the generator. There is no restriction so far, but the best results are achieved when the values of the layer range from -2 to 2. This interval is the most suitable since TerraSketch uses two constants. One is a constant restricting a range of vertical layer adjustment realized by parameter *offset* ranging from -1 to 1 and a constant for flat terrain profile scaling which is 0.2. Values outside the interval -2 to 2 might not work well with these constants and lead to unexpected results.

Now we can proceed to the next step by merging all layers together starting

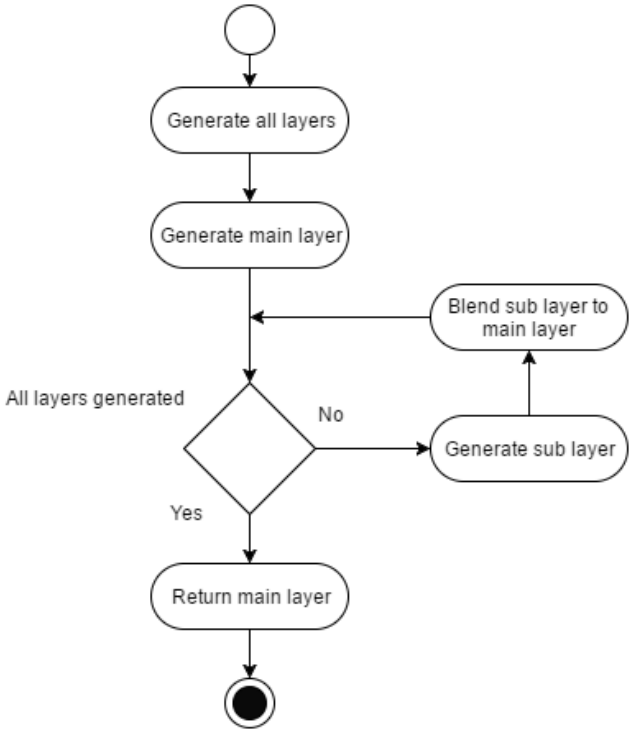


Figure 3.5: Generator flow

with the base layer. The base layer may not be present during generation since it is not mandatory. In such case an empty layer is created instead. The base layer uses the same generation process as any other layer. Having the base layer ready, one layer after another is positioned according to the position of the source polygon, because a generator generates only the smallest rectangular area possible.

When all layers are in the correct place we can take one after another and blend them to the base layer using their mask. The first thing is to use the mask to decide how much the pixel is visible and then blend it to the base layer using a selected blending function the result of which is then adjusted by adding an *offset* parameter changing the vertical position of the layer.

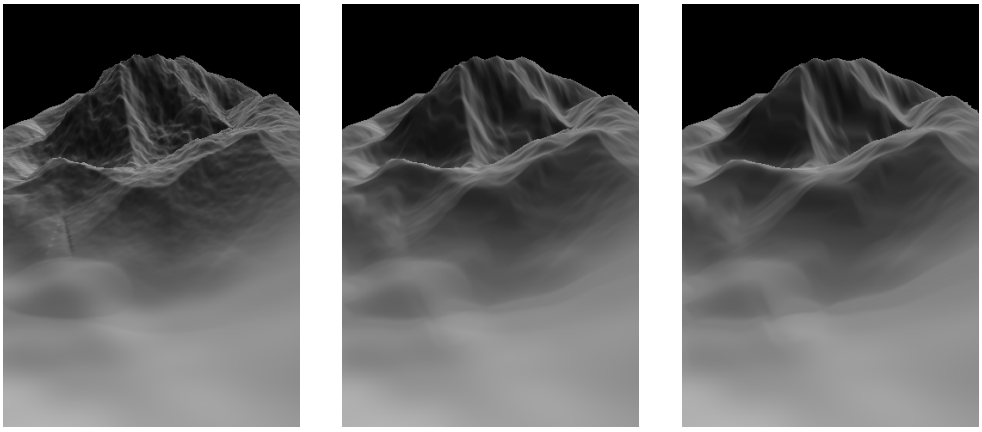
3.4 Erosion simulation and river generation

The final step before the terrain heightmap is done is to simulate erosion and create a river drainage map which is then applied on a terrain. The erosion is applied first, because otherwise strong erosion can dramatically change terrain shape affecting river basin.

The erosion is performed on a finished terrain only, its strength depends on input parameters that affect the amount of iteration performed ranging from 30 up to 150. This step requires a lot of time to finish since it iterates the whole heightmap many times.

A comparison of different erosion strengths is depicted in Figure 3.6. It is clear that erosion itself changed the bumpy noisy terrain seen in Figure 3.6a into a more naturally looking landscape depicted in Figure 3.6b or in Figure 3.6c.

The second step of a terrain post processing is to add rivers using the algorithm



(a) (b) (c)

Figure 3.6: Examples of different erosion strength settings: a) no erosion applied, b) 90 iteration of erosion applied, c) 150 iteration applied

described in the previous chapter. The river sources coordinates are randomly generated and each river is generated separately creating its own drainage map. All drainage maps are merged together. The merged map contains 1 on a coordinate if any of the source maps has a river on it and 0 otherwise. It is then scaled to $\frac{1}{100}$ and subtracted from the terrain which creates a river basin shown in Figure 3.7. This algorithm does not generate any confluences. Confluences can be detected during the merging process but when two rivers intersect, their basin from the intersection downstream is identical. This fact is given by the algorithm that is deterministic. The intersection detection is useful when the amount of joined river has any impact on its flow.

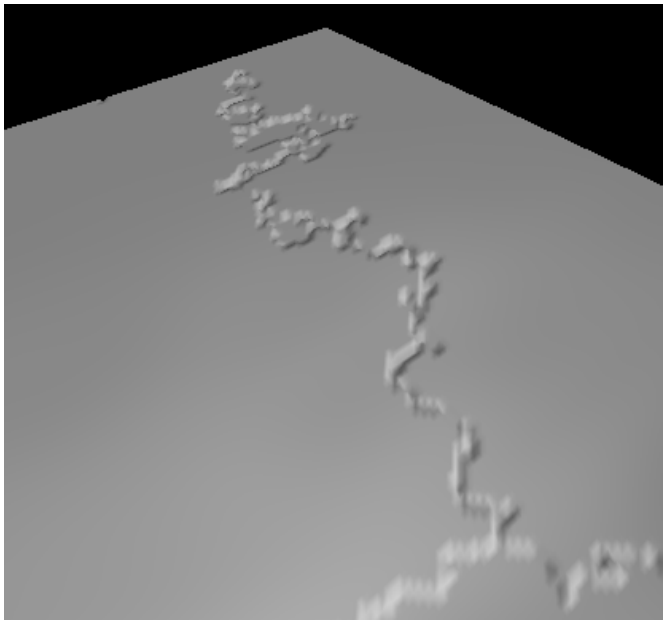
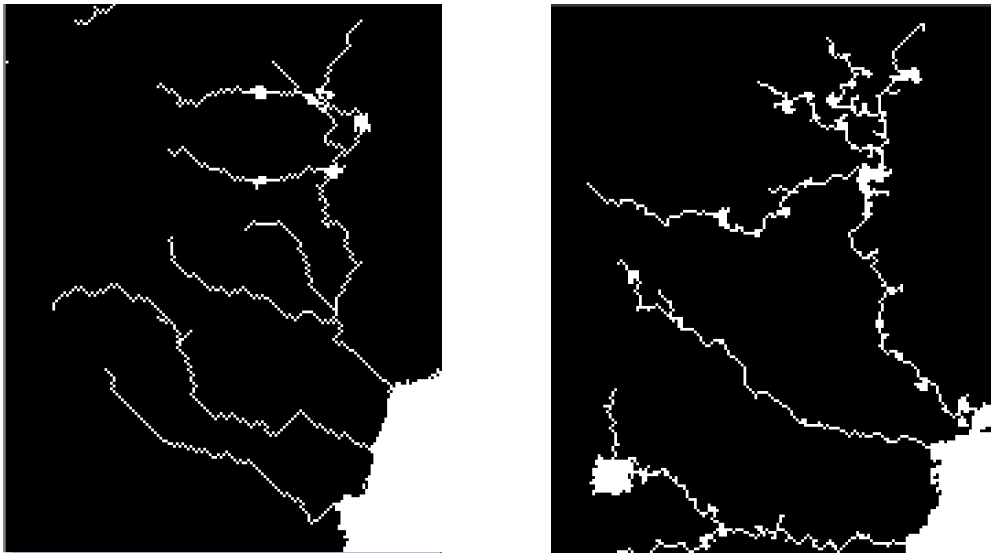


Figure 3.7: Rendered river

The result is shown in Figure 3.8a. We can see that the result is unacceptable for its straight lines. This problem is solved by temporarily adding noise to the heightmap. The lowest neighbor picking is then influenced by the noise which



(a) (b)

Figure 3.8: River drainage network: a) original output of the algorithm, b) the output with applied noise.

does not create straight lines on a place of rivers or shores (see in Figure 3.8b).

This solution has some drawbacks though. Noise is very soft but in some cases it can dramatically change the direction of the river flow. The source map M_s is smoother than the noised map M_n. When the algorithm finds a river for M_n, the river flow should have more interesting flow with meanders and other dramatical direction changes that were influenced by the noise. The details originating from the noise are not present on M_s, which can cause rivers to flow up to the hill in a very small scale and lake shores to copy the previous terrain M_n, which does not correspond to M_s on which the river is applied resulting in a surface that is not exactly flat.

3.5 Heightmap adjustments

When the heightmap is generated, a user can edit it directly using the tools that serve to minor terrain adjustments. The view features a canvas that allows the user to see and interact with the generated heightmap shown in a gray scale. The heightmap values are mapped to 8bit values with p_L set to 0 and p_H set to 255 using the formula:

$$p_{x,y} = (h_{x,y} - h_L)\frac{1}{h_H - h_L}. \tag{3.1}$$

Terrain details are edited with convolution brushes that have a circular shape only. A size of the circle is given by the *tool size* parameter which gives us the area of pixels affected by a kernel of the selected brush. The *tool size* does not affect the kernel size, a brush of a very large size has the same effect as the small one, but only on smaller areas.

The *fade* parameter changes the effect of a kernel on pixels that are close to the brush border. The *fade* parameter value ranges from 0 to 1 affecting the relative

$$\begin{matrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{matrix}$$

Table 3.4: Blur matrix

$$\begin{matrix} 0 & -\frac{1}{9} & 0 \\ -\frac{1}{9} & \frac{5}{9} & -\frac{1}{9} \\ 0 & -\frac{1}{9} & 0 \end{matrix}$$

Table 3.5: Sharpen matrix

distance where the effect of weakening the brush starts. The *strength* parameter affects opacity of the result of convolution. When the *strength* parameter is set to 0, kernel won't be applied and when it is 1, the result of convolution will completely replace the previous value. The implementation of brushes is described in Appendix A.

The brush effect is represented by a kernel of a fixed size. TerraSketch implements several types: *blur*, *sharpen*, *elevate* and *lower*. The blur tool uses a 3×3 matrix filled with ones shown in Table 3.4, which is known as a box blur algorithm. The *Sharpen* tool uses a matrix shown in Table 3.5. Working with this tool is tricky as it affects existing edges by multiplying their values.

The *lower* and *elevate* tools just slightly change the value of a pixel. The *elevate* tool adds a small constant to the pixel value while the *lower* tool does the exact opposite.

A brush is applied on a canvas when the left mouse button is held down. Every pixel that the mouse hovers above is affected by the brush. The mouse does not move continuously, which is a reason why we need to interpolate its position. The mouse movement creates a path that is sampled each k pixels and the brush is applied. Values computed by the brush are not applied directly to the heightmap, they are held in a temporary object until the mouse button is released.

3.6 Terrain preview

The last view shows the generated terrain and immediate preview changes made on the heightmap view. It allows a user to fly in space and observe the terrain 3D model. The model does not have textures and it is rendered gray and features shading only.

The terrain mesh is based on a heightmap each pixel of which on the x, y coordinate is translated to 3D where x, y corresponds to the pixel values and the z coordinate is a color of the pixel. Four pixels on the $(x, y), (x + 1, y), (x, y + 1), (x+1, y+1)$ coordinates taken from the heightmap form a square that consists of two right-angled triangles. One is created by $(x, y), (x+1, y+1), (x, y+1)$ and the other by $(x, y), (x+1, y), (x+1, y+1)$. Every pixel has its normal computed, which allows us to use shading.

A user can see the terrain using a standard first person camera. Using a mouse the user affects viewing direction and changes the camera position via keyboard. Camera implementation uses three orthogonal vectors. One is directional, which corresponds to the direction in which the user looks, the second is a vector of a camera's side and the last one is a cross product of the previous two pointing upwards. The camera does not allow a tilt so it is free to rotate in X and Y axes. For more details see Appendix A.6 for in-depth code documentation and Appendix B for a user guide describing its controls.

3.7 Common features

Any time during the work a user can save or load the project. When the terrain is satisfactory, the user can export it. The save file is in XML format that exactly reflects the data structure of TerraSketch. For a complete description of the format, see Appendix C.

Data is stored in a serializable tree structure. A root contains parameters of the world such as resolution, erosion strength, amount of rivers and parameters of its base layer, namely a profile, a level of detail, an offset and a blend mode. The root also contains a set of field descriptors that contain a polygon, order and also a profile,a level of detail and an offset.

The implementation does not create invalid states of the data structure that have to be checked before saving. The only requirement is that the polygon must be non self-intersecting, which is checked during the polygon definition and confirmation.

The final terrain can be exported to a very simple format, which is a text file with two values where the first means the real vertical position of the lowest point and the second is the position of the highest one. The rest of the file is a 2D array of the actual heightmap values ranging from 0 to 1.

4. Discussion and future work

This work presented a method for virtual terrain generation using polygonal sketching. The demonstrated process begins with a noise enhanced with fractional Brownian motion and turning it into a terrain using a detail enhancement, hydraulic erosion and river drainage simulation.

The user defines a terrain feature using a polygonal area, which results in visible straight edges seen on a generated layer. This problem can be solved by using curves instead of straight lines.

For the generation, we have chosen an older version of noise that has visible axis aligned artifacts that are partially removed by a fractional Brownian motion algorithm that create a decent base for further enhancement. The program design allows smooth transition to a modern type of noise, for example Simplex noise that does not produce the mentioned artifacts. Also, the process of generation uses generators that are currently not customizable by a user, which restricts the definition to a predefined terrain profiles only.

The most visible artifact appears during the layer masking process. Mask is created by a blur, which produces artifacts visible mainly on a layer with low detail settings. The source of this problem was not yet found and is a subject for further analysis.

We have experimented with more river generation simulations. The chosen one looks more realistic than the simulation proposed by Zelený [2009] even though it lacks more advanced features such as basin widening and its depth. We have also observed that a terrain without hydraulic erosion looks unnatural even though we have followed simplification of Olsen [2004] and simulated the process with natural looking results.

Now we have a solid base to build on featuring graphical user interface, framework with architecture allowing easy extensibility, but there is still a lot of work needed.

Future work

The general problem of TerraSketch is performance – the generation process is slow. More parallelization can be employed to speed it up by quicker algorithms and re-rendering can be sped up significantly by caching layers and rendering only changed ones. Also changing any parameter should redraw its respective layer.

The quality of a terrain can be improved by using another type of noise that does not have visible artifacts or implementing nonlinear stitching. The river drainage simulation should generate rivers the basin of which is widened when intersected by another river.

The future version of TerraSketch should be more customizable. A user should be allowed to customize a generator. The implementation will allow the user to insert custom noise functions as plugins, define complex generators using available components and influence the process in other ways such as defining heightmap pixel transforming functions or loading an existing heightmap as a layer.

Every editor that interacts with a user should be user-friendly. To improve

this in TerraSketch, we should implement more intuitive polygon definition tools and customizable view as well as more export formats.

To extend TerraSketch function we can implement more tools for the detail editor and interactive 3D view allowing a user to edit a terrain mesh. Also along with the terrain, a texture can be generated and mapped to 3D object as well.

Appendices

A Code guide

The code, save files and executable are distributed along with the work (see Attachments 1, 2 and 3). The code is written in C# 6.0 using Microsoft Visual Studio 2015. It requires the .NET framework 4.6 and Internet access to build properly. Internet access is required to download all nuget packages, which are just external libraries distributed on demand when the build starts. Every C# solution must have a startup project set, the user has to make sure that *TerraSketch.View* is selected.

There are also 3D party library dependencies that come with the need to render terrain in 3D. A terrain is rendered using primarily *OpenTk* to support *OpenGl* and *OpenTk.GLControl* library to access the 3D viewport from view and some secondary supporting libraries *AGG*, *NCalc*, *Trochuewitz* to support the random point generation and *MathHackers.Vector* math to support calculations. The libraries are distributed along with the source code.

A.1 Platform

TerraSketch is a software that interacts with a user using graphical user interface (GUI). GUI is implemented on an obsolete windows platform WinForms. The successor of WinForms GUI is Windows Presentation Foundation (WPF) which havs slightly different philosophy of usage and a completely different method of view definition.

Both platforms recommended usage is to create three layered architecture which consists of a model layer, view layer and presenter layer (MVP) or view model (MVVM). MVP and MVVM have only slight differences [1]. TerraSketch uses the MVP architecture that allows very quick upgrade from WinForms to WPF if needed. The most distinguishable feature of MVP is a two-way communication between a presenter and a view.

A.2 Structure

The application functionality is strictly separated into several projects. The MPV architecture is further divided into finer groups of projects by its purpose (as seen in the Table A.1). The Figure A.1 shows dependencies of the core projects.

The layer of presenters contains a presenter object for each view that contains all visible properties and action that the view performs. The properties visible on a view are bound to a view using standard windows binding. The action binding uses a custom made framework that for each action uses `Command` object and is stored in `TerraSketch.View.CommandBinder`. `Command` is an object that has two methods `Execute()` and `CanExecute()`. The `CanExecute()` method return information that says if a execute method is prepared and the Execute method

[1] https://www.scribd.com/document/291716836/MVVM-vs-MVP-vs-MVC

Layer	Purpose	Name and description
Model	Data	TerraSketch.DataObjects World and layer descriptors
		TerraSketch.DataObjects.Export Wrappers that alter xml serialization
		TerraSketch.DataObjects.SaveLoad Handles saving and loading
		Common.DataObjects Geometry data object and basic data structures
	Visualization	TerraSketch.Generators Takes care of generating separete layers
		TerraSketch.Heightmap.Composer Merges layers together
		TerraSketch.Layer Data structure representing layer
		TerraSketch.BitmapRendering Layer to bitmap converter
	Utilities	TerraSketch.Heightmap.Tools Utilities for layer transformations
		TerraSketch.FluentBuilders Api for simplier generator definition
		TerraSketch.Resources Structuralizes dataobjects
		TerraSketch.Logging Logs layers as images
		Common.MathUtils Math support
	Abstraction	TerraSketch.DataObjects.Abstract Interfaces for data objects
		TerraSketch.Generators,Abstract Iterfaces for generators
View	WinForms	TerraSketch.View Application startup project containing all forms
	Console	TerraSketch.SideDoorModule Allows to simply access and debug features
Presenter	2D	TerraSketch.Presenters Presenters and view interfaces
	3D	TerraSketch.VisualPresenters Presenters hadling 3D view
Tests	Unit testing	TerraSketch.Tests Test for a whole application
		Voronoi.Tests Test for Voronoi generator

Table A.1: Projects of TerraSketch

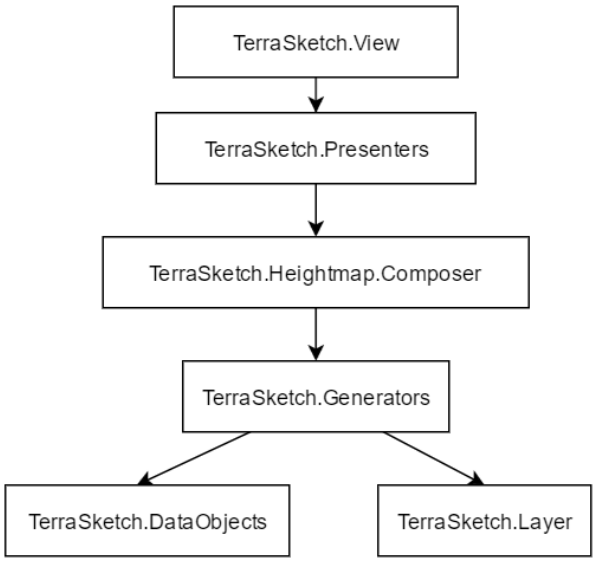


Figure A.1: Project dependencies

execute the action. Both method are customizable. Property binding detailed description can be found on a Microsoft developer websites [2]

A.3 Flow

The main application flow can be generalized in a following manner. The user sets properties on a presenter and then execute an action that uses them. For example, the purpose of a field view is to let a user define a data structure that is an input for a terrain generation. The user creates polygon using mouse events and sets their properties using the binding then the generation action can be executed. The purpose of the heightmap view is completely different but uses the same work flow.

The user interaction is solved using the binding mechanism which is the strongest feature of this platform. It also keep code clean and helps to separate view and presenter layer. Unfortunately, this is not true for the last view that still uses events to handle user interaction. Events are handled on a view that calls actions on a visual presenter. That is a reason why this presenter is in a separate project.

In a following text, we are going to focus on a process of a generation from its beginning. Parameters that affect a generation process are prepared in a compile time. The parameter detail is implemented as a command pattern Gamma et al. [1994] which means that it contains some behavior. An example of an application of this pattern is shown in the Figure A.2.

The second parameter that is ready in the compile time is a `Profile` that is implemented as an holder for a generator. This implementation requires no deciding in a generation process. Its behavior is inside input objects. For more details about parameter setup see a `DataSourcesMananger` in a TerraSketch.Resources project.

[2] http://tinyurl.com/msdn-notifications

```
public void Enhance(INoiseParametersDetails np)
{
    np.FromDepth = 4;
    np.ToDepth = 10;
    np.Amplitude = .40f;
    np.BaseAmplitude = 1.0f;
    np.Lacunarity = 2.0f;
}
```

Figure A.2: Example of function altering noise parameters.

A.4 Terrain generation

The terrain generation is performed by calling a Compose function with a world data structure on a `TerraSketch.Heightmap.Composer.HeightmapComposer` object. It runs a code that translates all fields, generate layers and merge them with a base layer A.3. A composed terrain heightmap is then drawn on a second view allowing user to edit detail with brushes.

A.5 Heightmap

This view allows to edit detail using brushes. The selected brush is applied on the view using MouseDown event on a `HeighmapPresenter`. When this action is detected a temporary 2D array with the same size as a heightmap is created by `BeginPaint()` method to store temporary values computed with a selected tool. Moving mouse with left button pressed calls `UpdatePaint` method that is updates a path of captured mouse locations. On all points of a heightmap in a selected distance from the path is a kernel of a brush applied and the result stored into temporary map. It is never applied again until the event stops by `FinishPaint()`, then the values from the temporary map are copied back to the heightmap.

A.6 3D view

TerraSketch uses *OpenGl* to render a terrain. It has to be initialized with shaders, materials and system informations before the first use. The initialization is performed in `Visualization3DPresenter.LoadOpenGL()` method. An input terrain heightmap must be converted to a mesh. This is covered by `TerraSketch.VisualPresenters.Visualization3DPresenter.Render()` method that is called periodically from a `OpenGl.GlControl` on a view.

The `GLControl` is a WinForms component that works as a view port. It is controlled using several events handling user inputs. A mouse position controls camera rotation and keyboard changes its position. Those changes requires a control to redraw using Paint event which calls the `Render()` method on a `VisualPresenter`.

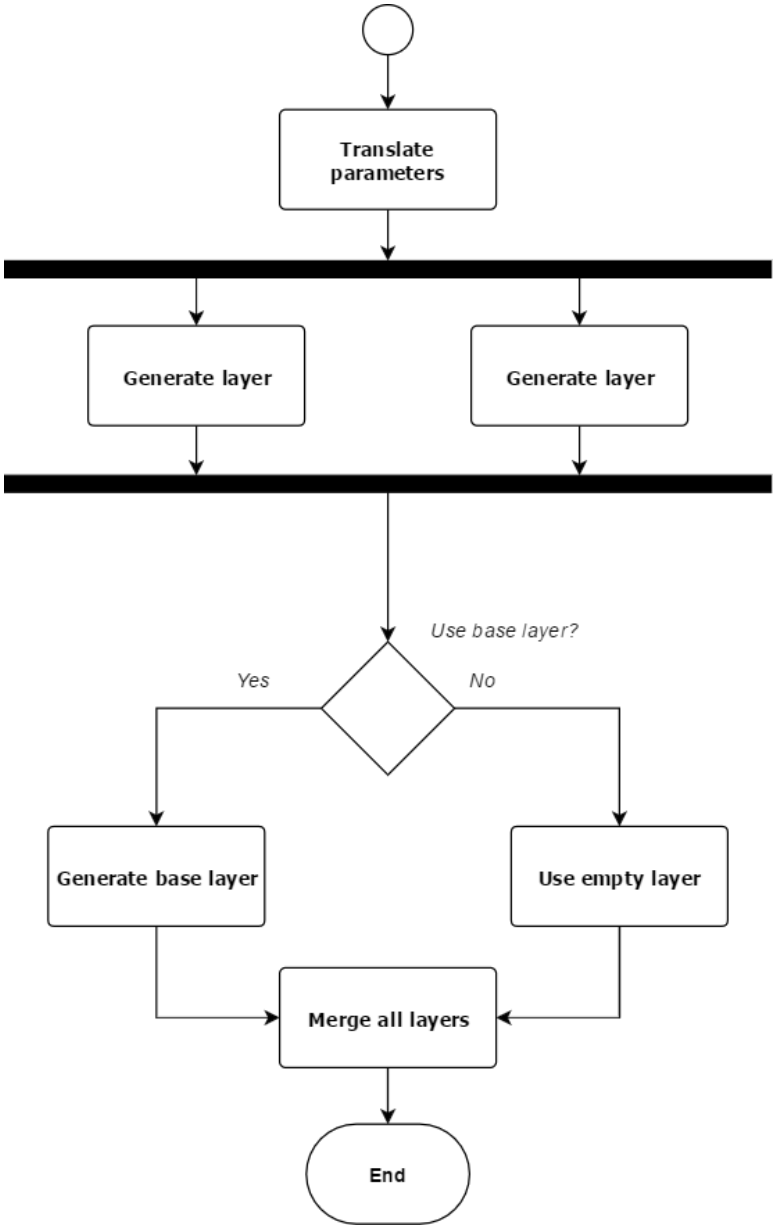


Figure A.3: Flow diagram of a composition process

A.7 Save/Load

The saving and loading implementation can be done easy with xml serialization and deserialization. Unfortunately a data structure does not support serialization because it uses dependency injection, has object with properties without setter and uses a lot of interfaces. Each object has to be copied to its serializable equivalent.

The coping process is customizable and allows to transform copied data. For example, saved file contains only caption of a selected profile even though we discussed that it contains a whole generator. The saving is easy but during a loading, we have to gather available profiles and find one whose name corresponds to the saved caption.

The whole saving and loading process can be simplified with non trivial effort by getting rid of readonly properties and dependency injection.

A.8 Logging

If a solution is built with a DEBUG symbol active, it produces debug images from various steps of algorithms. The results are stored in a *[Solution]/TerraSketch.View/bin/debug/debugImages* folder.

B User manual

B.1 Prerequisites and installation

The software requires

1. Windows
2. .Net framework 4.6
3. Mouse and keyboard

If all the prerequisites are met, extract all files from an archive ts.zip and double-click the TerraSketch.View.exe file which starts the application.

B.2 Structure

TerraSketch is a single window application with several tabs, each having a specific purpose. The tabs are: the *Field* tab that offers a canvas and a set of tools for polygon definition and global map settings, the *Heightmap* tab that shows the generated heightmap along with a set of edit tools and the last is the *Preview* tab with a 3D view port that shows the terrain in 3D.

Field view

Field view (shown in the Figure B.1) features a canvas on which a polygon should be defined. Polygon definition starts with a double-click anywhere on the canvas, which creates a square. The user can change the polygon with tools from a toolbox on the left side of the view. The toolbox consists of four separate groups of tools that help to define the area more easily. The groups contain tools for:

1. altering the order of layers,
2. zooming in or out,
3. polygon transformation,
4. polygon points selection and management.

A detailed description of each tool is shown in the Table B.1.

Each polygon has parameters altering generation of a feature it represents. The parameters with their effect are shown in the Table B.2. All parameters can be set on the top of the view. There are also three global parameters shown in the Table B.3 affecting the final terrain. The base layer parameters can be set after clicking the *Set Up Base* button. These parameters take effect when the *Use base* option is checked. When the terrain is set, the generation process can start by clicking the *Generate* button.

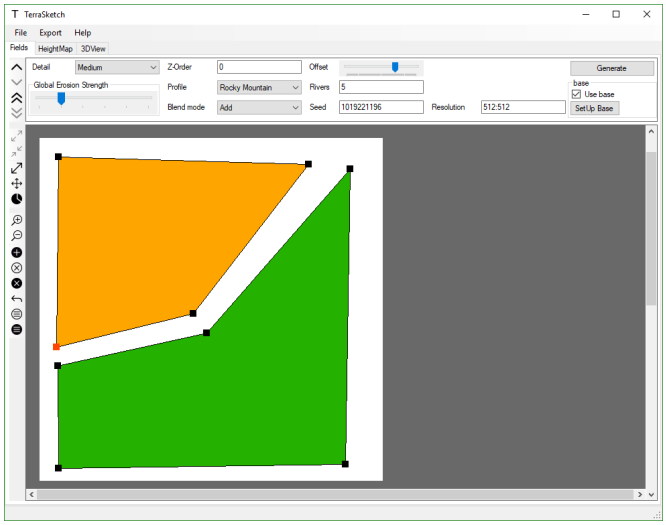


Figure B.1: A screenshot of Field view.

Heightmap view

The generated terrain can be altered in this view (shown in the Figure B.2) using convolution brushes. Select a brush in the combo box with the *Tool type* caption on the top of the view, set all the parameters such as *Tool size*, *fade* and *strength*. Brushes are applied by holding the left mouse button and moving a cursor. There is no history of changes, for that reason there is the *Recalculate heightmap* button recalculating the terrain from the ground and the *Clear heightmap* button deleting the heightmap from the data structure excluding it from the save file.

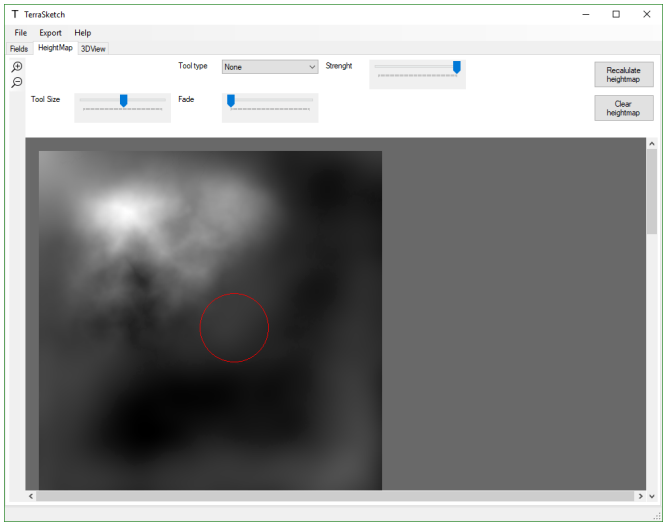


Figure B.2: A screenshot showing a heightmap

3D view

The generated terrain can be previewed on the last tab *3D Preview* (as shown in the Figure B.3) allowing the user to fly around using an FPS camera. Its controls are described in the Table B.4. Mouse sensitivity can be changed by moving the

Sensitivity slider and the position and rotation of the camera can be set to default values using the *Reset camera* button.

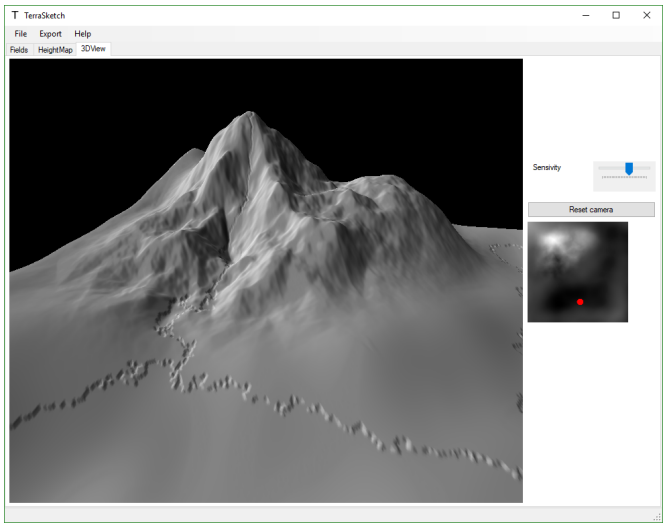


Figure B.3: A screenshot of the 3D preview.

Common features

The menu bar at the top of the window consists of three items. One is *File* that lets the user *Close*, *Save* or *Load* the file with a standard file dialog, the second is *Export* that exports the heightmap to a text file and the last one opens the *About* form with credits.

Icon	Description
	Brings a layer to the front
	Sends a layer back
	Brings a layer to the top
	Sends a layer to the bottom
	Subdivides selected edges
	Merges selected edges to one point in the center
	Scales selected points
	Translates selected edges
	Rotates selected points around their center of gravity
	Zoom out
	Zoom in
	Adds a new polygon
	Deletes selected point(s)
	Deletes a selected field
	Interrupts the current operation
	Selects all points of a polygon
	Deselects all points of a polygon

Table B.1: Tools altering the order of a layer

Name	Effect on the selected field
Level of detail	Changes the level of detail
Profile	Changes the profile
Z-order	Changes the order in which layers are merged
Blend mode	Changes the mode used in merging
Seed	Changes the seed of a random generator
Offset	Changes the vertical position of a terrain feature

Table B.2: Field parameters

Name	Effect on the terrain
Rivers	Amount of river sources spawned on the terrain
Resolution	Resolution in pixels affecting the layer size
Global erosion strength	Slider with a predefined strength of an erosion

Table B.3: Global parameters

Control	Action
W	Move forward
A	Move left
S	Move backward
D	Move forward
X	Move down
Z	Move up
Click + move	Rotate the camera

Table B.4: Camera controls

C Saved file format

The save file format is a XML. The structure of the whole file is shown in the Figure C.1. *Fields* and *BaseLayer* elements are omitted for simplification. The *World* element contains its own parameters, base layer and other fields. The *BaseLayer* and *Fields* elements have a structure described in the Figure C.2

```
<SaveItemXmlWrapper>
  <World>
    <Fields>...</Fields>
    ...
    <Fields>...</Fields>
    <WPar>
      <ResStr>512:512</ResStr>
      <ErosionStrength>0</ErosionStrength>
      <RiverAmount>0</RiverAmount>
    </WPar>
    <UseBase>false</UseBase>
    <BaseLayer>...</BaseLayer>
    <ExpPar>
      <MaxHeight>100</MaxHeight>
      <MinHeight>0</MinHeight>
    </ExpPar>
  </World>
</SaveItemXmlWrapper>
```

Figure C.1: Example of a save file structure without fields

```
<Fields> <!--or BaseLayer -->
    <ZOrd>1</ZOrd>
    <FPars>
        <Detail>
            <Caption>Low</Caption>
        </Detail>
        <Profile>
            <Caption>Canyon</Caption>
        </Profile>
        <BlendMode>
            <Caption>Replace</Caption>
        </BlendMode>
        <Offset>0</Offset>
        <Seed>213070490</Seed>
    </FPars>
    <Poly>
        <Pts>
            <X>189</X>
            <Y>316</Y>
        </Pts>
        <Pts>
            <X>229</X>
            <Y>316</Y>
        </Pts>
        <Pts>
            <X>229</X>
            <Y>356</Y>
        </Pts>
        <Pts>
            <X>189</X>
            <Y>356</Y>
        </Pts>
    </Poly>
</Fields>
```

Figure C.2: Example of a field or a base layer element structure

Bibliography

Franz Aurenhammer. *Voronoi diagrams and Delaunay triangulations*. World Scientific, Singapur u.a, 2013. ISBN 9789814447638.

Farès Belhadj and Pierre Audibert. Modeling landscapes with ridges and rivers. pages 151–154, 2005. doi: 10.1145/1101616.1101648. URL `http://doi.acm.org/10.1145/1101616.1101648`.

Berg. *Computational geometry : algorithms and applications*. Springer, Berlin New York, 2000. ISBN 3540656200.

J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Syst. J.*, 4(1):25–30, March 1965. ISSN 0018-8670. doi: 10.1147/sj.41.0025. URL `http://dx.doi.org/10.1147/sj.41.0025`.

John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1986.

I. Cojan and M. Renard. *Sedimentology*. Taylor & Francis, 2002. ISBN 9789058092656. URL `https://books.google.cz/books?id=eRwxDEjRwQcC`.

J.D. Foley. *Computer Graphics: Principles and Practice*. Addison-Wesley systems programming series. Addison-Wesley, 1996. ISBN 9780201848403. URL `https://books.google.cz/books?id=-4ngT05gmAQC`.

S Fortune. A sweepline algorithm for voronoi diagrams. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 313–322, New York, NY, USA, 1986. ACM. ISBN 0-89791-194-6. doi: 10.1145/10515.10549. URL `http://doi.acm.org/10.1145/10515.10549`.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994. ISBN 9780321700698. URL `https://books.google.cz/books?id=6oHuKQe3TjQC`.

Jean-David Génevaux, Éric Galin, Eric Guérin, Adrien Peytavie, and Bedrich Benes. Terrain generation using procedural models based on hydrology. *ACM Trans. Graph.*, 32(4):143:1–143:13, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461996. URL `http://doi.acm.org/10.1145/2461912.2461996`.

Adam R.C. Gritt. Master thesis on a procedural fish animation. Master thesis, Bournemouth University, Faculty of Computer Animation and Visual Effects, 2010.

Martin Kahoun. Procedural generation and realtime rendering of planetary bodies. Bachelors thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2010.

F. Kenton Musgrave. 16 - procedural fractal terrains. In David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, William R.

Mark, John C. Hart, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley, editors, *Texturing and Modeling (3)*, The Morgan Kaufmann Series in Computer Graphics, pages 488 – 506. Morgan Kaufmann, San Francisco, 3 edition, 2003a. ISBN 978-1-55860-848-1. doi: https://doi.org/10.1016/B978-155860848-1/50045-0. URL http://www.sciencedirect.com/science/article/pii/B9781558608481500450.

F. Kenton Musgrave. 20 - mojoworld: Building procedural planets. In David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, William R. Mark, John C. Hart, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley, editors, *Texturing and Modeling (3)*, The Morgan Kaufmann Series in Computer Graphics, pages 564 – 615. Morgan Kaufmann, San Francisco, 3 edition, 2003b. ISBN 978-1-55860-848-1. doi: https://doi.org/10.1016/B978-155860848-1/50049-8. URL http://www.sciencedirect.com/science/article/pii/B9781558608481500498.

Jacob Olsen. Realtime procedural terrain generation - realtime synthesis of eroded fractal terrain for use in computer games. Department of Mathematics And Computer Science (IMADA) University of Southern Denmark, 2004. Technical report.

Ian Parberry. Designer worlds: Procedural generation of infinite terrain from real-world elevation data. *Journal of Computer Graphics Techniques (JCGT)*, 3(1):74–85, March 2014. ISSN 2331-7418. URL http://jcgt.org/published/0003/01/04/.

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

Ken Perlin. 12 - noise, hypertexture, antialiasing, and gesture. In David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, William R. Mark, John C. Hart, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley, editors, *Texturing and Modeling (3)*, The Morgan Kaufmann Series in Computer Graphics, pages 336 – 410. Morgan Kaufmann, San Francisco, 3 edition, 2003. ISBN 978-1-55860-848-1. doi: http://doi.org/10.1016/B978-155860848-1/50041-3. URL http://www.sciencedirect.com/science/article/pii/B9781558608481500413.

Aristrid Lindenmayer Prusinkiewicz. *The Algorithmic Beauty of Plants*. Second printing. Springer-Verlag, Praha, 1996.

R.M. Smelik, T. Tutenel, K.J. de Kraker, and R. Bidarra. A declarative approach to procedural modeling of virtual worlds. *Computers and Graphics*, 35(2):352 – 363, 2011. ISSN 0097-8493. doi: https://doi.org/10.1016/j.cag.2010.11.011. URL http://www.sciencedirect.com/science/article/pii/S0097849310001809. Virtual Reality in BrazilVisual Computing in Biology and MedicineSemantic 3D media and contentCultural Heritage.

Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Declarative terrain modeling for military training games. *Int. J. Com-*

put. Games Technol., 2010:2:1–2:11, January 2010a. ISSN 1687-7047. doi: 10.1155/2010/360458. URL http://dx.doi.org/10.1155/2010/360458.

Ruben M. Smelik, Tim Tutenel, Klaas Jan de Kraker, and Rafael Bidarra. Interactive Creation of Virtual Worlds Using Procedural Sketching. In H. P. A. Lensch and S. Seipel, editors, *Eurographics 2010 - Short Papers*. The Eurographics Association, 2010b. doi: 10.2312/egsh.20101040.

Ruben M. Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes. A survey on procedural modelling for virtual worlds. *Computer Graphics Forum*, 33(6): 31–50, 2014. ISSN 1467-8659. doi: 10.1111/cgf.12276. URL http://dx.doi.org/10.1111/cgf.12276.

Irwin Sobel. History and definition of the so-called "sobel operator", 2014.

Shamus Young. Project octant, 2012. URL http://www.shamusyoung.com/twentysidedtale/?p=15742.

Matej Zábský. Geogen - scriptable generator of terrain height maps. Bachelor thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2011.

Jan Zelený. Rivers and lakes in artificial landscape. Faculty of Information Technology (FIT), Brno University of Technology, 2009. Technical report.

Attachments

1 Source code

An archive with a source code written in C# 6.0 in Visual studio 2015 on .NET 4.6. It contains a solution with projects and all files necessary for building.

2 Executable

An archive with an executable file along with its dlls.

3 Save files

An archive with the saved file ready to use.