ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/300253408

Noise Modeler: An Interactive Editor and Library for Procedural Terrains via Continuous Generation and...

Chapter · September 2015

DOI: 10.1007/978-3-319-24589-8_42

CITATIONS
0

READS
28

2 authors, including:

Anne C. Elster
Norwegian University of Science and Technology
70 PUBLICATIONS **285** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

CloudLightning View project

All content following this page was uploaded by Anne C. Elster on 06 July 2016.

The user has requested enhancement of the downloaded file. All in-text references underlined in blue are added to the original document and are linked to publications on ResearchGate, letting you access and read them immediately.

Real-Time Editing of Procedural Terrains

Johan K. Helsing and Anne C. Elster

Norwegian University of Science and Technology (NTNU), Trondheim, Norway
johanhelsing@gmail.com, and elster@idi.ntnu.no

Abstract. Procedural content generation is the act of creating video game content automatically, through algorithmic means. In online procedural generation, content is generated as the game is running on the consumers computer. Our Noise Modeler framework, is an approach to designing and modeling terrain for endless world creation. Noise and other functions are composited through a flow-graph editor similar to the ones used by procedural shader editors, and offline terrain generators. The framework enables non-programmers to edit models for procedural terrain while observing the effect of changes immediately in a real-time preview.

Keywords: Online terrain generation, noise synthesis, real-time procedural content generation, stochastic implicit surface modeling

1 Introduction

Realistic, detailed and interesting terrains can be very time-consuming to model manually. Procedural terrain generation tools generate terrains algorithmically, and may thus remove all limits of terrain size and detail (great for mobile devices and web applications). They also make it possible to feature a unique terrain each time an application is started (great for replayability).

Current tools for procedural terrain generation can be divided into two categories: **Stand-alone terrain editors** and **terrain generation libraries**. **Stand-alone terrain editors** save the terrains as non-procedural models that can be used by game engines. The tool itself, however, may not be included in the game. Hence, these terrains are essentially non-procedural when viewed by the end-user. **Terrain generation libraries** make it easier to write code that will generate terrains during run-time. However, they are very hard, to use for non-programmers, because they are aimed at game engine developers.

Since the majority of these libraries are written using single-threaded CPU code, using them may significantly increase the loading time of a game. Current procedural terrain editors only support offline generation of static geometry, typically a heightmap. This makes them unusable by games that require procedural content to be generated during run time. Little middleware for terrain generation exist, so many implement custom-made terrain solutions. Noise synthesis libraries such as **libnoise** may be used, but usually require a programmer to endure a cumbersome refine- recompile-rerun loop as part of the terrain design process. Additionally, few noise libraries support computation on the GPU [22].

2 Background and current approaches

Some recent games and their terrain models are shown in Table 1.

Table 1: Terrain models in some recent game engines.
Static/dynamic indicates whether terrains are editable during run-time.

Engine	Released	Heightmaps	Displacement	Smooth voxel	Languages
Upvoid Engine	2014	No	No	Yes, dynamic	C#
Unity 4.3	2014	Dynamic	No	With plug-ins	JavaScript, C#, Boo
Unreal Engine 4	2014	Static	No	No	C++, UnrealScript
CryENGINE 3	2009	Dynamic	No	Prior to 3.5.3	C++
Torque Game Engine	2007	Static	No	No	C++, TorqueScript
Source Engine	2004	Static	Static	No	C++, Lua, Python, ...
Panda 3D	2002	Dynamic	No	No	C++, Python

Heightmaps , the most popular terrain representation, uses two-dimensional grids of elevation data. Each value in the grid may be thought of as the height above sea-level for that particular grid location. Note that overhangs and caves are not supported, since each coordinate on the map may only have one single height value. Also, grids are evenly spaced, making it hard to support a varying level of detail in the model.

Vector displacement fields include the width and length directions in addition to the existing height offset. wheareas **Layered heightmaps** consists of multiple heightmaps where different materials of the terrain, such as sand, snow, gravel, and stone, each have a separate heightmap layer. The final height is a sum of all these layers. Terrains may also be represented by **3D meshes** and edited using polygonal modeling software (e.g. Blender, 3D Studio Max, or Autodesk Maya). They are typically used in game scenes where terrains are not an important part of the virtual world.

Voxel grids are three-dimensional grids of voxels. At each voxel coordinate, there may typically be air (nothing) or ground. A terrain surface can be approximated using a variant of the marching cubes algorithm [12, 7]; and does not have the topology constraints that heightmaps and vector displacement fields have. Caves with multiple exits, as well as floating islands, are perfectly possible. Examples of games and engines that use voxel terrains are: *Minecraft*, *Infiniminer*, *Upvoid Engine*, *Cube World*, and *Worms 4: Mayhem*.

Musgrave et al.[17] provides heightmap generating tools and 3D modeling tools, such as World Machine and Audodesk Maya that let the user combine noise and other signals through a visual node editor. However, the heightmaps have to be generated locally and stored as textures, and then shipped together with the game. Hence, to the end users, the content is static and many of the desirable properties of procedural generation are lost.

Procedural content generation (PCG) has been a common feature in games for a while, but has only the last decade generated academic interest. A useful introduction and overview of current research can be found in[21].

Terrain generation using **stochastic interpolation** generates terrains explicitly by evaluating a large batch of noise values at once. Mandelbrot [13] uses a two-dimensional **fractional Brownian motion (fBm)**, as an approximation of terrain altitudes. Fournier et al. [4] rendered terrains using **stochastic interpolation**, an approximation of **fBm** by recursively interpolating values with a pseudo-random offset proportional to the distance between the data points interpolated. Miller [16] describes the somewhat improved diamond-square algorithm, as well proposes a new — and slightly more complex — version without artifacts suffered byit, by sacrificing the requirement that the surface should have to pass through the control points.

However, when using these explicit techniques, one individual point cannot be generated without generating the rest of the model as well.

2.1 Implicit procedural techniques and noise

Implicit procedural techniques rely on a self-contained mathematical model to express the geometry of the terrain. Hence, arbitrary data points can be queried independently of the other points, and are thus often referred to as "point evaluation" [17]. Gamito and Musgrave [6] refers to this as stochastic implicit surface modeling.

A **procedural noise function**, may be used to approximate fBm. We used an unpredictable continuous function with range approximately $[-1, 1]$ and an approximate frequency of 1 Hz for this work. Ideally, the noise should also be isotropic, meaning that regardless of how you rotate the noise, it will look similar. A formal definition of procedural noise can be found in Lagae et al. [11].

On its own, noise is not a very close approximation to fBm, but by combining the function with itself scaled to different frequencies and amplitudes, it is possible to get an approximation satisfactory for terrain generation.

This implementation of fBm is a common choice among applications that need simple procedurally generated terrains. Babington [1] has for example used this implementation to generate terrains for the NTNU HPC-Lab snow simulator. Babington used the algorithm with Perlin noise as the noise function. Nordahl [18] enhanced this implementation by providing a GUI that could be used to adjust the inputs to the fBm function while the simulator was running. Musgrave [17] also contains a comprehensive guide on other ways noise can be used as a building block to create a wide range of natural structures.

Another approach to generating more natural terrains, is to start with a rough surface, for example the one produced by fBm, and simulate natural erosion processes. Hydraulic and thermal erosion are two such phenomenons. Our framework does not yet include erosion.

The libnoise *noise library* is widely used. Recently, it was used by Davis [2] to create a voxel terrain generator. The library is also used for many GUI applica-

tions for offline texture and heightmap generation. TerraNoise and Noise Mizer are such application However, they lack 3D previews of the terrain, showing only the heightmap texture.

Accidental Noise Library (ANL) , like libnoise allows the construction of a noise-based function through "modules" and "sources". In ANL, however, a module can answer queries about points in 2, 4 and 6-dimensional space in addition to 3-dimensional space.

The library also supports expressing functions in the scripting language Lua. These scripts may be parsed during run-time and used to generate terrain geometry online. It runs on the CPU only and does not take advantage of the computing power of GPUs.

GeoGen [25], is an open-source procedural heightmap generator that can be used for real-time generation. It is thus slow and not easily parallelizable. For example, Zábský [25] shows that generating an eroded 2048×2048 heightmap required over 7 minutes. In comparison, the GPU implementation of Olsen [19] required only 4 seconds on 7 years older hardware.

The software is licensed under GPLv2 and is therefore not usable for online generation by closed-source projects.

Procedural shader editors are used to design procedural textures, often called materials. While not commonly applied for generating terrains, use many of the same algorithms, and do use GPUs. are used. Noise synthesis is a common technique for both terrains and textures.

In fact it would be possible to design an online procedural terrain by creating a vertex shader and applying it to a tessellated plane. Or a fragment shader might be created that could be drawn on two flat triangles, and transferred back to the CPU in order to provide a heightmap for the game engine.

Many shader editors feature a flow-graph based editor, similar to World Machine. They are usually engine-specific. Only a few shader editors create shaders for multiple game engines, e.g. Allegorithmic Substance Designer Microsoft recently obtained a patent for a "visual shader designer" [14].

GPU implementations include Perlin's pixel shaders in [20], which used a hashing technique reliying on textures. He also sacrificed the fifth order interpolation polynomial in order to take advantage of hardware interpolation. Green [9] get identical results to the CPU version in [20] by storing the permutation vector in a 2D texture of height 1. McEwan et al. [15] uses a permutation polynomial instead of Perlin's permutation array. They also select gradients from the surface of a cross polytope surface instead of storing pre-computed gradients. Their implementation was twice as slow as a version that used texture lookups, but claim that the GPU's texture bandwidth tends to be a scarce resource, hence there is often an excess of unused computation power. Their implementation include both Perlin noise and simplex noise in 2, 3 and 4D.

Level-of-detail (LOD) algorithms are used to render huge terrains that still give an acceptable level of detail close to the observer.
Continuous distance-dependent level of detail, or **CDLOD** [23], a recent LOD-algorithm, divides terrain geometry into a quadtree of square terrain patches, similarly to how ROAM divides a terrain into a binary tree of triangles. A variant of CDLOD has been used by Babington [1] to improve the terrain rendering in the NTNU HPC-lab snow simulator.

3 Our Approach

We have developed a framework consisting of three parts:

- a serialization format for terrain generators,
- a library for modifying and evaluating the format and
- a graphical user interface for editing height-map terrain while displaying a real-time preview.

Our approach uses a noise synthesis approach to terrain generation as described by Ebert et al. [3]. Terrains are designed by combining noise and other functions through functional graphs that can be serialized as JSON-files consuming only a few kilobytes of storage. A serialized graph can later be evaluated by a game engine through the aforementioned library. The library will parse the JSON-file into an internal graph representation, which is subsequently used to generate GLSL-code which will execute the terrain function on the GPU.
Few existing tools are capable of visualizing terrains that can later be generated online while the game is running. The only tool capable of this known to the author, is GeoGen, mentioned earlier.
Our approach draws inspiration from existing offline procedural terrain generators that are successfully used in game development. An interface has been developed which bears a close resemblance to the flow-graph editor of World Machine, one of the most successful commercial terrain editing tools available. While World Machine generates terrains offline, our tool still retains the ability to generate terrains online.
The tool is also novel because it can help model non-terrain features as well, such as vegetation density, air humidity, and transitions between biomes.

3.1 Concepts

We use **Implicit procedural surface modeling** for representing the terrain. A terrain generator is simply a function definition for a function that can answer queries about a terrain.
Such a function may be described as a directed acyclic graph of other functions. Each node in the graph has a specific type, which represents a particular mathematical function or algorithm, specifically how its inputs are transformed to produce its outputs. Gamito [5] refers to this as a hypertexture hierarchy.

The focus of the GUI editor is to develop terrains that can be used with existing game engines with as few modifications or plug-ins to the game engine as possible. As most game engines support heightmap based terrains, the design of the GUI is centered around creating a height function $f(x, y)$. Such a function can easily be used to evaluate a patch of a heightmap terrain at an arbitrary resolution and scale making the terrain as portable as possible. This is also why real-time previews are only available for heightmap terrains. The editor is still perfectly capable of creating models for voxel terrains and vector field terrains as well, it is just not possible to preview them.

3.2 Comparison to libnoise and ANL concepts

While they may seem similar at first glance, the concepts described here are quite different from most existing graph-based noise-generating tools, such as ANL, libnoise, Lithosphere, and World Machine.
While libnoise and ANL also represent a generation function as a graph of modules, the assumptions these tools make about the resulting function is quite different. While edges in an ANL or libnoise graph correspond to function calls, edges in our approach correspond to outputs being assigned to inputs. In our approach, the nodes, or modules, themselves are the function calls, while modules in ANL represent callable functions.
While ANL and libnoise modules have hard-coded function signatures, our approach lets the signature of a module be decided dynamically at runtime. Depending on its module type, a module may have any number of inputs. Modules may even have multiple outputs, since the connections between inputs and outputs refer to the outputs individually, and not to the entire module. Being able to choose a function signature freely has several benefits. One such example is the generation of vector displacement terrains, which needs three separate floating point values for each lattice point. With our approach, a module could simply return a three-dimensional vector, while with libnoise or ANL, it is necessary to make three separate queries to three different modules. Consequently, this may cause expensive recalculation if the three modules have source modules in common. The issue may be solved by using special cache modules, which temporarily store the result of the most recent computation, but these modules must be inserted manually.

3.3 Framework requirements

How fast does our implementation need to be in order be considered fast enough? Swink [24] argues that < 50 ms response feels instantaneous, a 100 ms delay is noticeable, but ignorable, whereas > 200 ms response feels sluggish. Thus, ideally, the delay would be below 50 ms, but a delay of 100-200 ms is also acceptable.
The efficiency of the terrain generation sets a limit for how detailed and how large terrains it is possible to render within this time frame. The following

gives an estimate of how many points that must be generated in order to render a terrain.
Assuming the terrain is rendered using clipmapped LOD, the number of needed vertices can be estimated by the following equations:

$$v = lw^2 - (l-1)(\frac{w-1}{4}+1)^2 = lw^2 - \frac{1}{16}(l-1)(w+3)^2 \quad (1)$$

where l is the number of LOD-levels, w is the width of one LOD-level in number of vertices, and v is the number of vertices required. $(l-1)(\frac{w-1}{4}+1)^2$ is subtracted to avoid counting some vertices twice, as $(\frac{w-1}{4}+1)^2$ is the number of vertices that overlap between two adjacent LOD-levels. We can also correct for frustum culling using the following formula:

$$v_{visible} \approx \frac{\alpha}{2\pi} v \quad (2)$$

where α is the field of view of the projection matrix measured in radians.
In order to get an estimate for the number of vertices needed and the area covered, the clipmap parameters used in REDengine 3 for *The Witcher 3* [8] has been inserted into Eq. (1) and Eq. (2). Inserting $l = 5$, $w = 1025$, and $\alpha = 60\frac{\pi}{180} = \frac{\pi}{3}$, to the equations gives us $v = 4\,988\,929$ and $v_{visible} \approx 831\,488$. Note that depending on the rendering approach, it may still be needed to generate more than $v_{visible}$ height values since frustum culling may be performed at a later stage in the rendering pipeline.
This means that in order to render a real-time preview with a quality and render distance comparable to modern video games, it must be possible to generate around 500 000 to 1 000 000 height values in less than 200 ms.

4 Architecture

Our framework has been divided into three parts, as shown in Fig. 1.
Since the generation code used by game engines is now independent from the GUI code, it does not matter if the GUI uses dependencies that are unacceptable for game engines. This means we are free to use GUI toolkits, OpenGL and other heavy libraries in the GUI application. A more detailed overview of the architecture and all dependencies can be seen in Fig. 2.
The rationale for also factoring out the serialization format, is that for some game engines, it might be impractical or impossible to use the library. I.e. it may not be possible for some scripting languages to call C++ functions, or wrap them in a language usable by the scripting language. By keeping the definition of the serialization format open and explicit, it is possible for developers to create their own code for parsing terrains that works on their deployment platform, while still having the benefit to be able to use our GUI application for modeling the terrains.

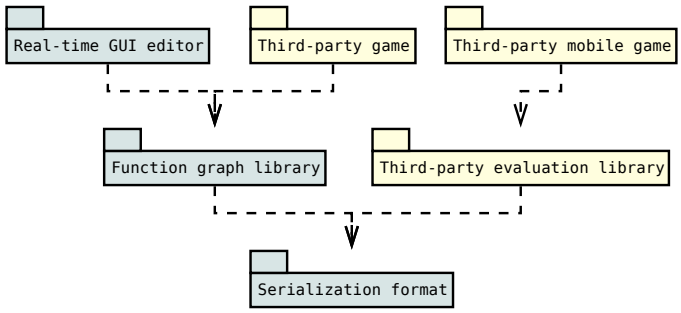

Fig. 1: Architecture overview. The arrows show dependencies. Blue boxes are part of the framework, while yellow boxes show potential third-party software.

4.1 Library architecture

The responsibility of the library is to provide an API for parsing, serializing, evaluating and modifying function graphs.
The library needs to interface with two stakeholders, the GUI application, and the game engines that use the framework.
For game engines, it is important that it is easy to interface with the library, and that the library is small, self-contained and runs on many platforms. As game engines are commonly implemented in C++, it will be easiest if our library is implemented in C or C++ as well. Also, it is usually fairly simple to create wrappers for C and C++ libraries in order to interface with them in engines that use other languages for extensions.
The terrain editor application on the other hand, has a different set of requirements. Most importantly, the requirements for interactive performance are much stronger. While game engines can often afford to wait a couple of seconds during a loading phase, the terrain editor needs to generate enough of a terrain for a high quality preview in under 200 ms. In order to achieve this kind of performance, it is crucial that the algorithm is implemented on the GPU. This is explained in Section 4.3.
The library has been carefully implemented without dependencies on the GUI application. This means that the library can be used by game engines to parse and evaluate functions described by our format without pulling in dependencies that are large, do not run on particular platforms, or have restrictive licenses (such as Qt). This also allows the source code of the library to stay relatively small and concise (approximately 3000 lines of code, avoiding the extra 4000 lines of code needed for the GUI application).

model is the most important part of the library. All other parts of the library depend on it. The model provides an object-oriented representation of

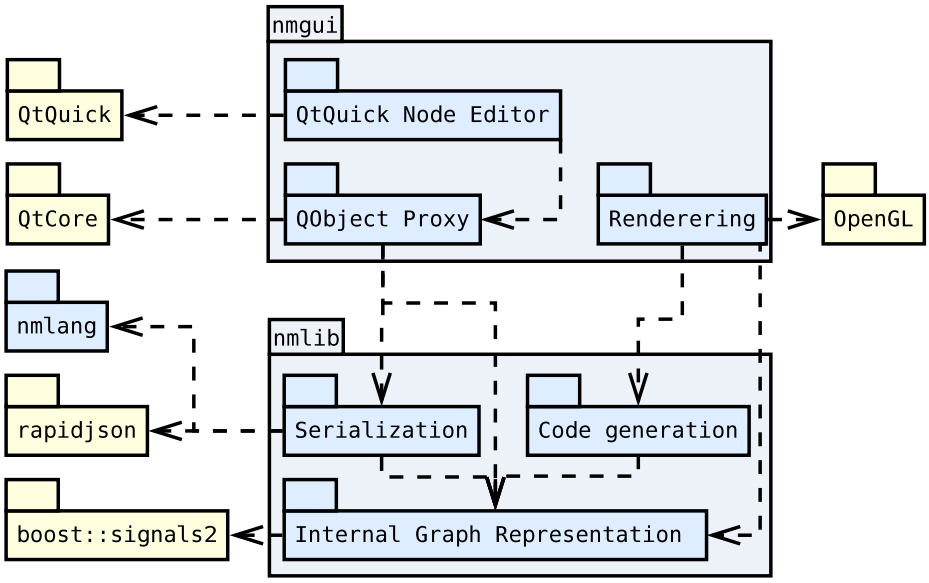

Fig. 2: Detailed framework architecture including dependencies and package modules. The arrows indicate dependencies.

function graphs and their relationships. The interface provides ways to modify and create new graphs.

serialization is responsible for serializing and parsing graphs (in the model) to and from JSON.

code generation is responsible for generating GLSL functions equivalent to the function graphs.

4.2 The Noise Modeler application

The GUI application will be referred to as the "Noise Modeler". We want the editor to run on numerous hardware and software configurations. To make this easy, the cross-platform GUI framework Qt and QtQuick were chosen. The core of Qt is also written in C++, and this makes it easy to interface with nmlib. QtQuick, however, can only use C++ classes that derive QObjects. To use QtQuick, we needed to wrap all classes in the model and serialization modules of nmlib. Although this involved an amount of tedious manual work, it meant that it was possible to take advantage of QML, which is a powerful declarative language for creating user interfaces.
The user interface of the application is described in Helsing [10].

4.3 Parallel computation of implicit terrains

All the CPU-based implementations of implicit stochastic terrain generation are too inefficient for our requirements.

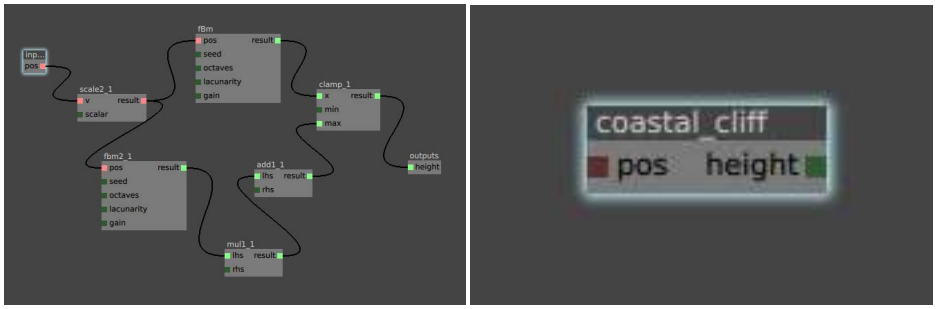

Fig. 3: Encapsulate complicated sub-graphs as new module types

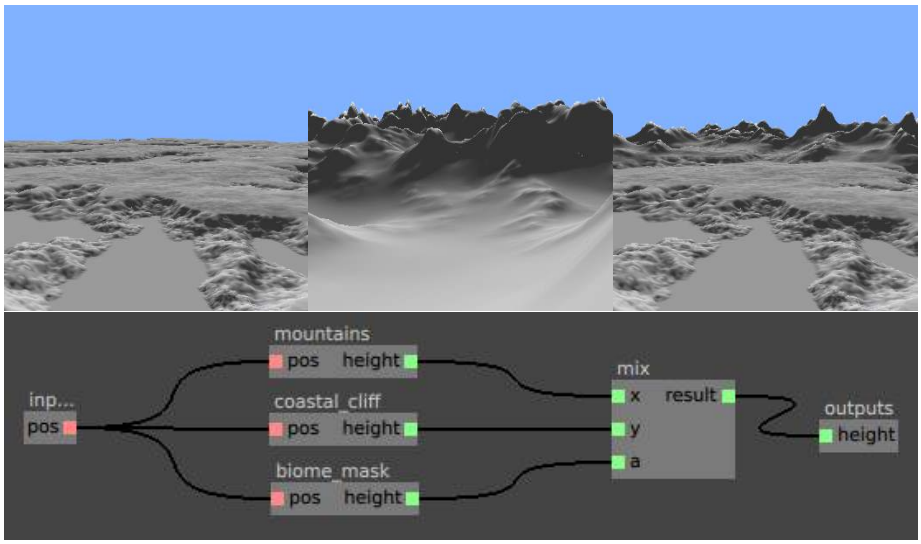

Fig. 4: Different biomes can be combined using a high-level mask, such as fractional-brownian motion with a low frequency and few octaves.

Batch computation of points on a stochastic implicit surface is clearly a problem that scales well on massively parallel architectures, such as GPUs. Furthermore, there are generally few branch instructions within the computation of a point, making the problem a perfect fit for the SIMD architecture of GPUs.
For these reasons, the GPU was an obvious choice of evaluation platform for our library. By choosing to evaluate terrains on the GPU, it becomes necessary to select a GPGPU API. Of numerous alternatives, OpenGL 3.0 was chosen. Other APIs were considered as well, including CUDA, Direct3D, OpenCL, and Mantle. The reason for choosing OpenGL, is that our library is primarily meant as middleware for the video game industry. Video game developers usually prefer to reach the widest audience possible. This has several implications for our requirements when choosing a GPGPU platform. It is desirable to:

- Support as many hardware configurations as possible: This rules out CUDA, as it is only available for NVIDIA GPUs, and Mantle, because it is only available for AMD GPUs.
- Support as many software configurations as possible: This rules out Direct3D, as it is only available for Microsoft's operating systems (Windows and Xbox).

Direct3D is partially supported by other operating systems through the Wine compatibility layer. Although this could potentially increase the number of supported platforms, using Wine may cause a significant performance impact. It is also a rather large software package, and may be seen as an unreasonable requirement to play a video game.
If elevation data is cached, it may also seem advantageous that computation of terrain data happen in the same framework that will use it for rendering, to avoid unnecessary data transfers and duplication. However, it is perfectly possible to share buffers between OpenCL and OpenGL, so this is not an important advantage.
While a definition of a stochastic implicit surface may be written by hand using GLSL, this is the very same approach often taken by game developers that we are trying to avoid. In our framework, the terrain model is built using a run-time editable model of functional composition. This model resides in CPU memory and is editable through a C++API. In order to bring computation to the GPU, the CPU library generates GLSL shader code for computing the terrain by traversing the graph model of the function.
The generated code is stand-alone, meaning it does not rely on any textures or other buffers to compute the function values. This makes it callable from a wide range of shader stages, including fragment, vertex, tessellation and compute shaders. It also has the added benefit that the portability requirement is easier to satisfy, since this limited set of GLSL functionality is easily portable to most platforms.
By generating GLSL code during run-time, a strictly serial part is added to the algorithm. Not only does the code have to be generated from the model, but

the GLSL shader also has to be compiled. This overhead only has to be executed once each time the terrain function changes. For example, if two patches of terrain are computed, this setup only needs to be performed once for the first patch. When a terrain is edited interactively, however, it means that it changes continuously and this overhead has to be executed after each change. The performance impact of this step is therefore significantly large, since the steps also require compilation of a GLSL shader, which could be expensive depending on the OpenGL implementation. This step has been benchmarked in Helsing [10] **??** for several OpenGL implementations.

4.4 Testing and verification

Because there was only one developer working on the project, sub-tasks and planning were managed by a simple project backlog on Trello.com.
Tests were written prior to implementing functionality, using the unit testing framework Google Test. Having these tests proved very useful during the iterative development of the framework. Some unit tests were also written for the rendering parts of the GUI application.
The main pilot tester was a friend of the developer working on an open-source game. His feedback helped shape the user interface of the application and fix oversights by the developer. Aside from this, the design of the interface also relies heavily on familiar user interface elements from similar applications, and this also lowers the need for user testing.
Our library demonstrates our approach makes it possible to generate changing terrains at interactive rates. This has been demonstrated in two ways: Firstly, by running the editor itself, it could be observed that terrains can be edited interactively. Secondly, a more thorough benchmark were performed as described in the following section.

5 Conclusion

Most terrain editors focus on offline procedural terrains, ignoring the powerful capabilities of procedural generation, the most important being replayability and vastness. Our GPU-based Noise Modeler framework shows it is possible to model procedural terrains in *real-time* in a user-friendly application, while at the same time retaining the ability to integrate with a game engine and generate terrains during run-time. The efficient computation of height values allowed a *real time* high-quality terrain preview to be updated with a vertex count comparable to state-of-the-art video games.
Our framework may be used to model heightmap terrains, and supports several popular algorithms for stochastic implicit terrains. Our framework may also model and generate — although not preview — other types of terrains, including voxel terrains and vector displacement terrains. It may thus integrate well with a variety of game engines with different approaches to terrain modeling.

Future work include adding support for previewing additional types of terrain representations such as voxel terrain and vector displacement terrain. In addition an effort to port the library to javascript has been started with the intention of making the format usable by web applications using WebGL.

References

[1] Kjetil Babington. "Terrain Rendering Techniques for the HPC-Lab Snow Simulator". Master's Thesis. Norwegian University of Science and Technology, 2012.

[2] Bjorn Curt Davis. "Terrain Generation Engine Using Voxels". MA thesis. California State University, 2013.

[3] David S Ebert et al. *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[4] Alain Fournier, Don Fussell, and Loren Carpenter. "Computer Rendering of Stochastic Models". In: *Communications of the ACM* 25.6 (1982), pp. 371–384.

[5] Manuel Gamito. "Techniques for Stochastic Implicit Surface Modelling and Rendering". PhD thesis. England: University of Sheffield, 2009.

[6] Manuel N Gamito and F Kenton Musgrave. "Procedural Landscapes with Overhangs". In: *10th Portuguese Computer Graphics Meeting*. Vol. 2. 2001.

[7] Ryan Geiss. "Generating Complex Procedural Terrains Using the GPU". In: *GPU Gems 3*. Addison-Wesley Professional, 2007, pp. 7–37.

[8] Marcin Gollent. "Landscape Creation and Rendering in REDengine 3". Game Developers Conference 2014. 2014. URL: `http://twvideo01.ubm-us.net/o1/vault/GDC2014/Presentations/Gollent_Marcin_Landscape_Creation_and.pdf` (visited on 07/09/2014).

[9] Simon Green. "Implementing Improved Perlin Noise". In: *GPU Gems 2*. Addison-Wesley Professional, 2005, pp. 409–416.

[10] Johan K. Helsing. "Framework for Real-Time Editing of Endless procedural Terrains". Master's Thesis. Norwegian University of Science and Technology, 2014.

[11] Ares Lagae et al. "State of the Art in Procedural Noise Functions". In: *Eurographics 2010-State of the Art Reports* (2010).

[12] William E Lorensen and Harvey E Cline. "Marching Cubes: A High Resolution 3D Surface Construction Algorithm". In: *ACM SIGGRAPH Computer Graphics*. Vol. 21. 4. ACM. 1987, pp. 163–169.

[13] Bernard Mandelbrot. *The Fractal Geometry of Nature*. CA: Freeman, 1982.

[14] S. Marison et al. *Visual shader designer*. US Patent App. 13/227,498. 2013. URL: `http://www.google.com/patents/US20130063460` (visited on 04/28/2014).

[15] Ian McEwan et al. “Efficient computational noise in GLSL”. In: *Journal of Graphics Tools* 16.2 (2012), pp. 85–94.

[16] Gavin SP Miller. “The Definition and Rendering of Terrain Maps”. In: *ACM SIGGRAPH Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 39–48.

[17] F Kenton Musgrave. “Procedural Fractal Terrains”. In: *Texturing & Modeling: A Procedural Approach*. Morgan Kaufmann, 2003.

[18] Andreas Nordahl. “Enhancing the HPC-Lab Snow Simulator with More Realistic Terrains and Other Interactive Features”. Master’s Thesis. Norwegian University of Science and Technology, 2013.

[19] Jacob Olsen. *Realtime Procedural Terrain Generation*. Tech. rep. University of Southern Denmark, 2004.

[20] Ken Perlin. “Improving noise”. In: *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. ACM. 2002, pp. 681–682.

[21] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014.

[22] Ruben M Smelik et al. “A Survey on Procedural Modelling for Virtual Worlds”. In: *Computer Graphics Forum*. Wiley Online Library. 2014.

[23] Filip Strugar. “Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD)”. In: *Journal of Graphics, GPU, and Game Tools* 14.4 (2009), pp. 57–74.

[24] Steve Swink. *Game Feel: A Game Designer’s Guide to Virtual Sensation*. Taylor & Francis US, 2009.

[25] Matěj Zábský. *GeoGen — Scriptable generator of terrain height maps*. Bachelor thesis. Charles University in Prague. 2011.

View publication stats