Framework for Real-Time Editing of Endless Procedural Terrains

Johan Klokkhammer
Helsing

Master of Science in Computer Science
Submission date: July 2014
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem description

Game developers commonly re-implement terrain generating code for each project they are working on, thus using up valuable development time. Furthermore, terrain generating code is often written in compiled languages, and this usually requires the code to be recompiled and the application relaunched each time a change has been made to the generator. Games and other virtual environments often use terrains generated using the implicit procedural techniques of Perlin and Musgrave.

These techniques have also been applied to heightmap-generating software and 3D-modeling software. Such programs commonly offer a node editor letting the user visually edit an expression tree consisting of common procedural algorithms and mathematical functions. The results of the changes can often be viewed in real time. These packages, however, do not let you run the generation on the client. A heightmap has to be generated in its entirety first; and can then be imported by the game or application as a non-procedural model. Hence, many of the desirable properties of procedural generation are lost in the process, such as the ability to generate infinite worlds or a new world each time a game is started.

The goal of this project is to combine the ease of use of visually editing an expression tree in a node editor with the ability to generate terrain on demand at the player's computer. An open-source library capable of serializing and deserializing graphs of functions will be developed, as well as a visual node editor interfacing with the library. The node editor should be capable of displaying generated terrains in real time. This will enable non-programmers to edit terrain efficiently, while retaining the ability to perform generation in-game. A reference plug-in for a real-time engine will be developed to demonstrate the potential of the framework. Existing game engines as well as the NTNU HPC-Lab Snow Simulator will be considered as candidates for the reference plug-in.

Assignment given: 15 January 2014
Advisor: Anne C. Elster, IDI, NTNU

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 consumer's computer.

Online procedural generation of terrains has become an important feature in many recent video games. The technique enhances the replayability and vastness of virtual worlds by offering a unique and endless terrain for each play session.

Procedural terrain generation is commonly achieved through noise synthesis. Adding, multiplying and filtering layers of noise at different frequencies and amplitudes, can lead to complex and realistic terrain models. This process of filtering and combining noise and other functions to create a final terrain function is usually done by issuing calls to a noise generating library in a programming or scripting language, such as C++ or Lua.

The process requires a programmer to write code, compile the code, run the program, observe the results, and then start over editing the code. This is a tedious, non-intuitive and time-consuming design process. Consequently, game designers are often forced to accept sub-optimal results because of time constraints or lack of control over the generation process.

Our framework, Noise Modeler, consists of a GUI application and a library for modeling terrains for endless-world creation. In this project, noise and other functions are composited through a visual flow-graph editor similar to the ones used by procedural shader editors and offline terrain generators. This novel framework enables non-programmers to edit models for procedural heightmap terrains while observing the effect of changes immediately in a real-time preview.

Designed terrains can be serialized to human-readable text files, consuming only a few kilobytes. By using our library, a game engine can load these text files in order to generate terrain data on-demand on the GPU.

To our knowledge the Noise Modeler framework is therefore unique in its cause. It may be limited in features, and rough around the edges in terms of usability, but it clearly outperforms existing noise libraries, and has terrain specific features and heightmap previews not present in procedural shader editors.

Sammendrag

Prosedyrisk innholdsgenerering (procedural content generation) er et begrep for det å generere spillinnhold automatisk ved hjelp av algoritmer. I online prosedyrisk generering genereres innholdet mens spillet kjører på spillerens datamaskin.

Online prosedyrisk generering av terreng har blitt en viktig del av mange nylige dataspill. Teknikken kan forlenge levetiden og størrelsen til et spill ved å tilby en unik og uendelig virtuell verden hver gang spillet startes.

Prosedyrisk terrenggenerering oppnås vanligvis ved å syntetisere og filtrere syntetisk støy. Ved å legge sammen, multiplisere og kombinere støy ved forskjellige frekvenser og amplituder, er det mulig å lage komplekse og realistiske terrengmodeller. Denne prosessen — bestående av filtrering og kombinering av støy og andre funksjoner for å lage det endelige terrenget — blir ofte opnådd ved hjelp av funksjonskall mot et bibliotek for syntetisering av støy gjennom et programmerings- eller script-språk.

Prosessen krever at en programmerer skriver kode, kompilerer koden, kjører programmet, observerer resultatet, og deretter redigerer kode igjen. Dette er tidkrevende, tungvindt, og ikke intuitivt. Resultatet er at spilldesignere ofte blir tvunget til å godta terrengmodeller de ikke er helt fornøyd med grunnet tidsnød, eller manglende kontroll over genereringsprossessen.

Vårt rammeverk, Noise Modeler, består av et grafisk brukergrensesnitt og et bibliotek for modellering av terreng for uendelige prosedyriske verdener. I dette prosjektet blir støy og andre funksjoner kombinert gjennom et visuelt redigeringsverktøy for grafer ikke ulikt tilsvarende verktøy for design av prosedyriske teksturer. Dette nye rammeverket gjør det mulig for ikke-programmerere å lage modeller for prosedyrisk generering av høydedata-terreng. Endringer på modellen kan observeres i en forhåndsvisning som oppdateres i sanntid.

Designede terreng kan serialiseres til et tekstformat som er forståelig for mennesker. Formatet bruker kun et par kilobyte for å representere et terreng. Ved å bruke vårt bibliotek, kan en spillmotor laste disse tekstfilene for å generere terrengdata når det etterspørres. Genereringen blir utført ved hjelp av klientens grafikkprosessor.

Noise modeler, er derfor — så vidt vi vet — et unikt verktøy. Det mangler riktignok en del funksjoner, og har et brukergrensesnitt som trenger justeringer i forhold til brukervennlighet, men rammeverket er klart raskere enn eksisterende støygenererende biblioteker og har terrengspesifikke funksjoner man ikke finner i prosedyriske shader-verktøy.

Acknowledgments

I wish to thank my supervisor Dr. Anne C. Elster for her valuable assistance and advice during the project.

Gratitude is also given to Colt McAnlis for letting me use one of his screenshots of a vector displacement terrain in this thesis, and Philip Trettner for his screenshot of the Upvoid Engine.

I would also like to thank NTNU and NVIDIA's CUDA Research Center and NVIDIA CUDA Teaching Center program for their contributions to the IDI/NTNU HPC-Lab.

Finally, I wish to thank Hanna H. Kamperud for her support throughout my study.

Contents

List of Tables

List of Figures

Listings

Glossary

ANL Accidental Noise Library, a library for compositing procedural noise.

API Application Programming Interface.

Biome Climatically and geographically contiguous areas with similar climatic conditions.

CUDA Compute Unified Device Architecture, a parallel computing platform developed by NVIDIA for their graphics processing units.

DAG A Directed Acyclic Graph.

fBm Fractional Brownian Motion.

Flow graph A directed acyclic graph, describing how outputs of algorithms flow to the inputs of other algorithms, in order to compute a final result.

Functional composition Combining the outputs of one or more simple functions as the inputs of another function.

Genotype An unexpanded procedural model. May be considered as a blueprint for the expanded model, the phenotype. For example, the parameter values and seed of a procedural algorithm can be considered a genotype.

GLSL OpenGL Shading Language. A programming language for the programmable parts of the OpenGL rendering pipeline.

GPL GNU General Public License. A copyleft free software license.

GPU Graphics Processing Unit.

GPGPU programming General Purpose Graphics Processing Unit programming. Using GPU hardware for non-graphical applications.

Heightmap A heightmap is a two-dimensional grid of elevation data.

JSON JavaScript Object Notation, a lightweight data-interchange format.

LGPL GNU Lesser General Public License. A copyleft free software license that allows dynamic linking with software not release under the LGPL/GPL licenses.

LOD algorithm Level-Of-Detail algorithm, an algorithm for choosing the appropriate level-of-detail of a 3D model, often based on factors such as screen resolution and the observer's position

MIT License A permissive free software license.

Noise When not specified otherwise, noise refers to procedural gradient noise, such as Perlin or Simplex noise, which are continuous functions varying between -1 and 1 in a seemingly random way.

Offline procedural content generation Content generation is performed as a part of the development of the software.

Online procedural content generation Content generation is performed during run-time.

OpenGL Open Graphics Library, a cross-language, cross-platform API for rendering graphics.

PCG Procedural Content Generation refers to creating game content automatically, through algorithmic means.

Phenotype The result of expanding the genotype. The output of a procedural algorithm. In the context of terrain, this might be a heightmap.

QML Qt Modeling Language, a JavaScript-based declarative user interface markup language for the Qt framework.

Qt A cross platform GUI application framework.

SIMD architecture Single Instruction Multiple Data architecture is a type of processor architecture that lets one operation be executed on multiple different data values at once.

Tessellation Subdivision of polygons into renderable primitives suitable for rendering, usually triangles.

Texture splatting A method for combining different textures by applying an alphamap that contains transparency values.

White noise A random signal with a probability distribution of zero mean and finite variance.

zlib License A permissive free software license.

Chapter 1

Introduction

A realistic, detailed and interesting terrain is often an important part of video games and virtual worlds, but can be very time-consuming to model manually. Procedural terrain generation tools aim to automate creation of terrains by generating them algorithmically. If applied carefully, these techniques may remove all limits of terrain size and detail, and may even make it possible to feature a unique terrain each time an application is started.

Current tools for procedural terrain generation can be divided into two categories: Stand-alone terrain editors and terrain generation libraries. Stand-alone terrain editors can be used to generate terrains on the game developer's machine. Such editors can 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 are middleware that make it easier to write code that will generate terrain during run-time. The problem with these libraries, however, is that they are very hard, or impossible, to use for non-programmers, because they are aimed at game engine developers.

Although many terrain generation techniques are highly parallelizable, the majority of these libraries are written using single-threaded CPU code. Generating a terrain may therefore take several minutes and can significantly increase the loading time of a game.

In this project, we look at how these two types of tools can be combined to create a user-friendly tool for procedural terrain generation that will allow efficient generation while a game is being played.

Such a tool would help reduce development costs of games with endless procedural terrains. Furthermore, a responsive and user-friendly tool can also remove restrictions on the creativity of game developers, and increase the quality of procedural worlds.

This chapter describes the benefits of procedural terrain generation (Section 1.1), a brief overview of how it is currently used in the game development process (Section 1.2), what the goal of this project is (Section 1.3), a list of research questions (Section 1.4) as well as an outline of the remaining chapters (Section 1.5).

1.1 Benefits of generating terrains procedurally

In the last decade, procedurally generated terrains have become an important part of many games and other virtual environments. There are a number of benefits from generating terrains procedurally as opposed to designing them by hand:

Shorter development time Creating a terrain by hand is very time-consuming. If parts of the terrain, or terrain details can be generated algorithmically it may cut development costs dramatically.

Increased detail Procedural generation of terrains allows an infinite amount of detail that would otherwise be impossible to implement due to storage space requirements.

Increased size As mentioned above, procedural generation may be used as a form of data compression. This makes it possible to increase the size of the virtual world to virtually infinity. The game *Minecraft*, for instance, features game worlds that are 16 times the size of the Earth.

Reduced application size On mobile devices, data usage is often an important limitation for developers. Many procedural algorithms require only a seed and a few parameter values for storage. This makes it possible to bring vast and detailed game worlds to devices with limited storage and bandwidth. Another use may be to reduce download times for custom maps in multiplayer games. This is often called database amplification.

Replayability By seeding the pseudo-random generator with a unique seed each time the game is started, a different world may be generated for each session, increasing the replay value of the game.

1.2 Current approaches

One of the most prominent types of terrain generation is based on the implicit modeling techniques described by Perlin and Musgrave [1]–[3]. Gamito [4] refers to these techniques as stochastic implicit surfaces. Through functional composition, noise and other mathematical functions are combined to create pseudo-random, endless terrains. Game developers frequently implement these techniques from the ground up for each project they are working on, or they may utilize one of the noise generation libraries available.

Developers usually write terrain generating code directly in C++ or their programming language of choice. After each change, the code has to be recompiled and the application relaunched before the results can be observed. This often leads to a delay of considerable length before the effects of a change can be assessed. Additionally, computation that could have been offloaded to the GPU may still be executed on the CPU because of game developers' unfamiliarity with GPGPU programming or because of limited development time.

The techniques of Musgrave et al. have already been applied to heightmap generating tools and 3D modeling tools, such as World Machine [5] and Audodesk Maya. These programs often let the user combine noise and other signals in a graph through a visual node editor. This lets the user design terrains efficiently and preview the effect of the changes after a short delay. The problem with these applications, however, is that it is not possible to run the generation online[1] at the end-users' computers. Using these tools, heightmaps have to be generated in their entirety at the game developer's computer, before they are 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 (see Section 2.1).

1.3 Thesis goals

The goal of this thesis is to develop a technique that combines the ease of editing terrains using a visual node editor with the ability to generate terrains in-game in real time. An open-source library capable of serializing and de-serializing graphs of functions will be developed, as well as a visual node editor interfacing with the library. The node editor should be capable of displaying generated terrains in real time. This will enable non-programmers to edit terrains efficiently, while retaining the advantages of performing generation online.

A reference usage of the middleware will be developed in order to demonstrate the potential of the framework. Plug-ins for existing game engines as well as the HPC-Lab snow simulator at NTNU may be considered as suitable targets.

[1]In this context, online refers to generating the content at run-time, and is not related to the Internet. See Section 2.1.

1.4 Research questions

Another goal of the thesis is to answer the following research questions:

RQ1 How can functional composition for stochastic implicit terrain surfaces be expressed in a portable, platform-independent manner?

RQ2 Suppose a terrain is represented by functional composition and is editable during runtime. How can efficient terrain calculation be implemented to support a three-dimensional preview reflecting terrain changes at interactive rates?

RQ3 Can a single approach be used express procedural terrain models in multiple terrain representation paradigms?

1.5 Thesis outline

Chapter 2 gives an introduction to background knowledge related to this project.

Chapter 3 explains how the problem and research questions were approached. The fundamental concepts of our approach are presented, and the software architecture of the framework is outlined.

Chapter 4 explains the details of how the solution was implemented.

Chapter 5 contains a discussion of the resulting software package, and to which extent it solves the problem and fulfills the requirements. Results from benchmarks are also presented.

Chapter 6 gives a conclusion of the project, discusses its shortcomings and how they may be fixed. Several possibilities for future research projects are also proposed.

Bibliography contains the referenced formatted according to the IEEE standard.

Appendix A is a user's guide for the developed software.

Appendix B is a single-page poster for the project.

Appendix C lists code for the benchmarks in Section 5.1

Appendix D contains an API reference for the library.

Chapter 2

Background

In order to create a useful tool for terrain generation, background knowledge of several topics is required. Most importantly, it is necessary to understand the topic of terrain generation, but other topics are important as well.

Before procedural terrain generation is discussed, it is useful to have a basic understanding of the area of procedural content generation and the terminology used in this context. This will be discussed in Section 2.1.

Terrains can be represented in many ways. Choosing a certain terrain representation model will result in different trade-offs between supported topologies and performance. In Section 2.2, several models for representing terrains will be presented.

In Section 2.3, we will give an account of what fractal terrains are, and how noise can be synthesized to create such terrains. These techniques have several common applications in game development, and we will give an overview of the two most common uses, noise libraries (Section 2.5) and offline procedural terrain generators (Section 2.6). Procedural shader editors will also be discussed in section Section 2.8, since they share many techniques and algorithms with terrain generation. Some of these tools may even be used to generate terrains.

A real-time procedural terrain editor needs to be able to render terrains in such a way that it will look similar to how a game engine would render it. Section 2.9 offers an overview of OpenGL, the library that was used for rendering, while Section 2.10 contains an explanation of widely used and state-of-the-art algorithms for real-time terrain rendering.

In the last section, Qt, the GUI application framework used by the editor implementation, is presented (Section 2.11).

2.1 Procedural content generation

Procedural content generation (PCG) is the act of creating game content automatically, through algorithmic means [6].

Procedural content generation is closely related to the term procedural generation (note the absence of the word "content"). Procedural generation is a wider term which includes concepts such as dynamic light maps and procedural textures. A requirement for PCG is often described as algorithms that affect gameplay substantially. Consequently, although both procedural textures and procedurally generated heightmaps share many of the same techniques and algorithms, only terrain generation is usually regarded as PCG. However, if the generated terrain is used solely for cosmetic purposes, such as backgrounds far off in the distance, it can be argued that it is not truly PCG.

While procedural content generation has been a common feature in games for quite a long time [6]–[8], there has not been much academic interest in the subject until the last decade [6], [8]. Although not yet been published, the first textbook on the subject has recently been written by Shaker, Togelius, and Nelson [8], and a draft is available for download[1]. The textbook serves as a useful introduction and overview of current research on PCG.

Togelius, Yannakakis, Stanley, *et al.* [6], and later Shaker, Togelius, and Nelson [8], provide a useful taxonomy for PCG which will be used to explain how the approach of this project relates to previous research and existing PCG tools. A short summary of the most relevant parts of the taxonomy will be given in this section.

Online versus offline One of the most important distinctions in PCG is the one between online and offline generation. Online generation means that content is generated during run-time, while the player is playing the game. Offline PCG, on the other hand, is generation of content during development time or right before the player starts a game session. The taxonomy of Togelius, Yannakakis, Stanley, *et al.* [6] does not make a distinction between **offline** generation taking place at the game developer's computer and **offline** generation happening at the player's computer. In this thesis, these distinctions will be referred to as development-time and run-time offline generation respectively. Run-time procedural generation will also be used to refer to both online and offline generation taking place at the end-user's computer.

This is the aspect in which the goal of this project differs the most from existing terrain generation tools. Most of the benefits of procedural generation listed in Section 1.1 only apply when terrains are generated online. For instance,

[1]http://pcgbook.com

development-time procedural generation can not be used to increase detail, size or enhance replayability of a game.

Another feature, exclusive to online PCG, is the possibility to create player-adapted content. For instance, Valve's first-person shooter Left 4 Dead analyzes player behavior in order to create a specialized experience for each player [8].

Necessary versus optional Necessary content is content that is required for the game to be playable. Optional content may for example be side-quests or add-ons to weapons. There are often a big differences in requirements for quality for necessary and optional content.

Terrain generation may be used for both extremes, but it usually falls within the necessary spectrum.

Random seeds versus parameter vectors All procedural algorithms expand content based on some sort of compact representation. Some algorithms generate content based on one single seed value, while others require additional parameters to specify the properties of the content. This axis of having no inputs, and having many inputs, is also commonly referred to as the number of degrees of control.

Stochastic versus deterministic If a procedural algorithm relies upon a seeded random number generator, it is considered a stochastic algorithm. Deterministic algorithms, on the other hand, can not be seeded and given the same inputs, they always produce the same output.

Constructive versus generate-and-test A constructive algorithm generates the content once, and is then finished. A generate-and-test algorithm, on the other hand, generates the content, and then tests whether it meets certain criteria. If the test fails, the content is discarded and regenerated. This is repeated until satisfactory content is generated.

Automatic generation versus mixed authorship In automatic generation algorithms, the input from game designers are limited to tweaking algorithm parameters. In mixed authorship algorithms, the designer is more involved in the process, perhaps by providing a rough sketch which can be expanded by the algorithm. Smelik, Tutenel, Kraker, *et al.* [7]'s SketchaWorld is an example of this.

Genotypes and phenotypes Genotypes and phenotypes are concepts borrowed from the field of genetics. A genotype corresponds to an organism's DNA while a phenotype corresponds to the actual observed properties of the animal.

When content is generated procedurally, it is often generated from a small set of input data. This data set is referred to as the genotype. A genotype may be expanded into a phenotype by running a procedural algorithm. The phenotype is a complete model, ready to be presented as content.

In the context of terrain generation, a genotype may refer to the input parameters to an algorithm, while the corresponding phenotype is the output produced by executing the algorithm with those inputs.

2.2 Terrain models

In order to discuss methods for generating terrain data, it is important to investigate how terrains are usually represented and used within game engines. Following are presentations of several models and how they set limits for supported geometry and performance.

2.2.1 Heightmaps

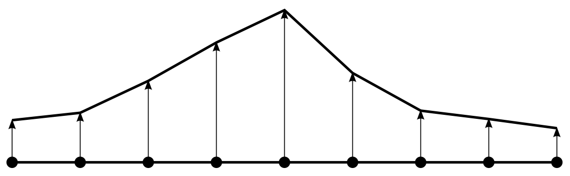


Figure 2.1: Heightmap terrains

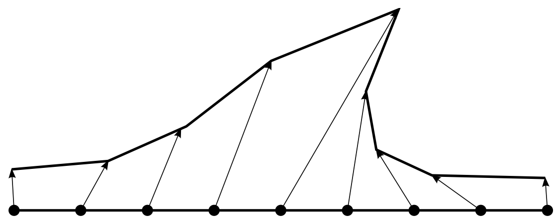
A heightmap, or heightfield, is a two-dimensional grid of elevation data. The index of each cell within the map corresponds to a point in the virtual world. The value stored at the cell is the height offset of that particular point. Each value in the grid may be thought of as the height above sea-level for that particular grid location. A 2D simplification of heightmap a terrain can be seen in Fig. 2.1.

Heightmaps are probably the oldest and most widespread representation of terrain data. They can be stored very efficiently, since only one floating point value is needed for each point on the map. Due to their wide-spread usage, much effort has also been put into developing efficient algorithms for rendering heightmap terrains. Several such algorithms are presented in Section 2.10.

Although very popular, heightmaps impose several restrictions on the terrain geometry. Overhangs and caves are not supported by heightmaps, 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.

Despite these limitations, heightmaps are still widely used even by modern game engines [9]–[13].

2.2.2 Vector displacement fields


(a) 2D-simplification of vector displacement

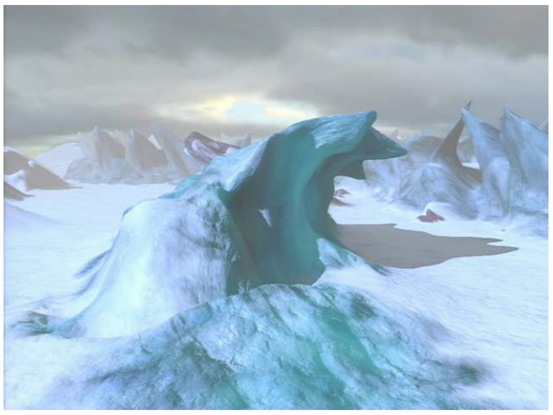

(b) Screenshot of the Halo Wars terrain engine

Figure 2.2: Vector displacement terrains

The vector displacement field is an extension of the heightmap model. It aims to solve several shortcomings of heightmaps, like support for overhangs and a varying level of detail.

A vector field adds two additional attributes to each grid cell in the map. The attributes describe offsets in width and length directions in addition to the existing height offset [4], [14]. Each vertex can now have an arbitrary three-dimensional offset from its original position in the grid. This allows a vertex to be offset so that it has the same length and width coordinate as another vertex, thereby creating an overhang. This feature is illustrated in Fig. 2.2.

This technique also allows vertices to be moved from areas which require few vertices (e.g. flat areas with little detail) to areas which require a higher level of detail.

It should be noted that vector displacement field terrains have the same topology as a heightmap terrains. This means that many of the same algorithms (i.e. level-of-detail algorithms) can be used for vector displacement fields as for heightmaps. It also means that caves with more than one entrance may not be represented using the model since that would change the topology of the model.

Vector displacement fields were used in the game Halo Wars developed by Ensemble Studios. The technique has been described in detail by McAnlis [14].

McAnlis [14] also converted displacement field terrains to heightmap terrains in order to interface with some of their AI systems. This was done by rendering the terrain from an orthographic top-down perspective into a framebuffer, and then using the depth coordinates as heightmap values. This approach may be used when overhangs are mostly cosmetic and not important for gameplay.

2.2.3 Layered heightmap

A layered heightmap consists of multiple heightmaps. 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. This representation allows more sophisticated erosion models, and can also be used to select which texture to use at a given point by selecting according to the topmost non-zero layer [15]. This can also aid other procedural algorithms, such as those for vegetation distribution.

If the terrain is destructible, layered heightmaps can be used to keep deformation data separate from the original terrain data [9], [12].

2.2.4 3D meshes

In some use cases, terrains are designed by editing the geometry of a 3D model, using polygonal modeling software. Applications such as Blender, 3D Studio Max, or Autodesk Maya may be used for this purpose. This technique is very flexible with regard to the topology of the terrain. Making overhangs and caves is trivial. The drawback of this technique is that it may not work well with some level-of-detail algorithms, as it may not be straightforward to automate creation of models with varying level of detail. It may also be difficult to respond to queries about the height of the terrain at a certain location. This can make it difficult to integrate the terrain with other systems that assume this functionality. AI and path-finding are examples of such systems.

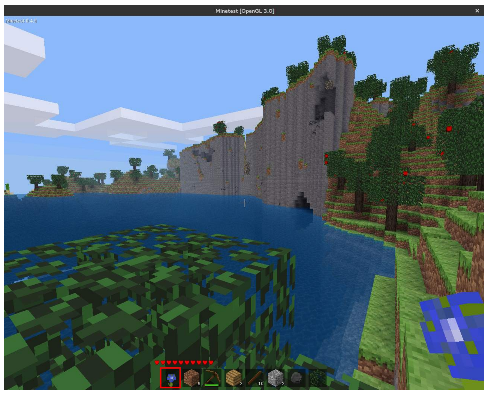
This approach is most commonly used in games where terrains are not an important part of the virtual world. For example, games that take place in a city may use a 3D mesh to describe a small patch of terrain in a park. The reason for this is that other parts of such games often have complex geometrical models, and modeling the terrain with the same system may be convenient.

2.2.5 Voxel grid

A voxel grid is a three-dimensional grid 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 [16], [17]; an example of this can be seen in Fig. 2.3b.

A voxel grid representation of a terrain 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.


(a) *Minetest*, an open-source game with a block-like terrain

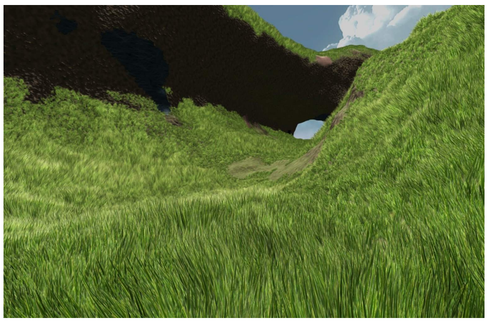

(b) Upvoid Engine, a more realistic implementation with a smooth voxel terrain.

Figure 2.3: Two types of voxel terrains.

Voxel grids have often been dismissed in game development because of the excessive disk space needed to save the grid. With procedural algorithms, however, this issue is circumvented since voxel data can be computed during run-time.

Voxel data is often generated using a terrain density function.

Examples of games and engines that use voxel terrains are: *Minecraft*, *Infiniminer*, Upvoid Engine, *Cube World*, and *Worms 4: Mayhem*.

2.2.6 Overview of common game engines and their supported terrain models

Table 2.1 contains a list of some of the most widely used game engines, and which terrain models they support by default. Note that although some engines may be limited to heightmap terrains, low-resolution voxel terrains are usually possible to implement by instancing large cubes using one of the engine's supported scripting languages. Many engines also have plug-ins making this process easier.

As we can see, heightmaps is clearly the most widespread model among recent game engines. Unity, CryEngine, and Panda3D natively support dynamic heightmaps that can be changed at run-time. This means that these engines can take advantage of an online procedural terrain generator.

Although vector field displacement is a promising technology, it does not seem to be widely adopted. Valve's Source Engine was one of the few engines supporting this, but the geometry is fixed at development-time.

Many engines do not allow changes to the terrain during run-time. The reason for this, is that light maps are often precomputed at development-time in order to increase performance during run-time. Although this has been a useful tech-

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

Table 2.1: Some game engines and their supported terrain models. Static/dynamic indicates whether terrains are editable during run-time. All engines support block-like voxel terrains through their scripting languages.

nique to increase performance, it puts restrictions on the features of the game engine. Many new games advertise a dynamic lighting model, since this will allow deformable terrains, or dynamic day and night cycles [12], [18]. It could be argued that since precomputed lighting is incompatible with these features, it will be eventually be phased out.

Unity 4

Unity, sometimes also called Unity3D, is a game engine commonly used by independent game developers, and smaller projects.

The game engine has built-in support for static heightmap terrains which can be edited with various brushes in an integrated terrain editor. It is also capable of importing elevation data saved as 16-bit grayscale RAW files.

Game programming is done through scripts written in C#, JavaScript or Boo. It is possible to change the terrain heightmap through these languages using the function:

```
void TerrainData.SetHeights(int xBase, int yBase, float[,] ↩
    heights);
```

This function takes a two-dimensional array of height data as input and overwrites the existing terrain. The array must be located in CPU memory. Another similar function is available for setting splat values, “SetAlphamaps”.[2]

Since Unity supports interfacing with C code through C#, terrain generating code written in C or C++ may be wrapped using C# and this function.

Unity is also extendible, and there are numerous plug-ins for voxel terrains available. One such plug-in, “Uniblocks”, allows game developers to use their own terrain generation platform[3].

[2] https://docs.unity3d.com/Documentation/ScriptReference/TerrainData.html
[3] https://www.assetstore.unity3d.com/en/#!/content/14768

Unreal Engine 4

Unreal Engine is a game engine used by many high-budget, graphically intensive games.

Like Unity, heightmap terrains are supported and can be edited using an integrated editor which can also import textures [11].

Terrain data for the built-in “Landscape” module may not be generated at the client during run-time, because of the use of a technique often called baked (or precomputed). Before a distributable game is created, the Unreal Engine editor calculates light contribution from static lights, assuming the world geometry is static. This means that if the terrain had been editable, then the lighting would have had to be recomputed, and currently this is only possible through the editor that runs on the game developer’s machine.

It should, however, be possible to circumvent the issue by using a module called “FDynamicMeshBuilder” to generate custom geometry during run-time. The source code for the engine is also available to paying subscribers, so it might be possible to modify the engine itself to use only dynamic lighting.

CryENGINE 3

CryENGINE 3 is a game engine targeting graphically intensive games. Game programming can be done by writing C++ code against the CryENGINE API.

The engine supports heightmap terrains by default. Since CryENGINE 3 uses dynamic lighting, it is not required to pre-compute parts of the lighting [18]. This means terrain data can safely be modified during run-time using the `void ITerrain::SetTerrainElevation` function.

2.3 Fractal terrains

Mandelbrot [19] proposed using fractals as a basis for terrain synthesis, claiming terrains exhibit a self-similarity of features on both a large and a small scale. Specifically he suggested that a two-dimensional fractional Brownian motion (fBm), a kind of fractal noise, would make good approximation of terrain altitudes. The idea caught on quickly, and soon many renderings of fractal landscapes appeared in scientific papers.

2.3.1 Stochastic interpolation

Fournier, Fussell, and Carpenter [20] rendered terrains using an approximation of fractional Brownian motion by applying a technique they called “stochastic interpolation”. The technique consists of recursively interpolating values with a pseudo-random offset proportional to the distance between the data points interpolated.

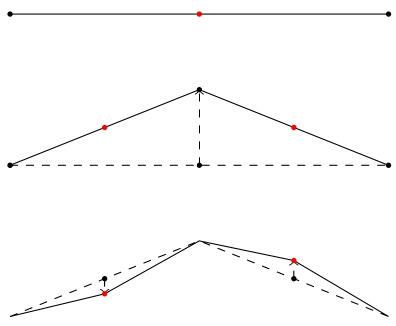


Figure 2.4: Two iterations of one-dimensional midpoint displacement

One algorithm using the technique, often called the midpoint displacement algorithm, is illustrated in Fig. 2.4 and works like this:

1. Start with the four corners of a terrain patch.
2. The point in the middle of all four corners is calculated as the average of all four corners, plus an offset that is proportional to the size of the terrain patch.
3. Calculate the midpoints on each edge as the average of the two neighboring corners.
4. Subdivide the patch into four smaller terrain patches, and execute recursively for each patch.

The is flawed, however. As pointed out by Miller [21], the resulting terrain will have axis-aligned ridges since the most dramatic offsets will always happen on the same points in the grid. For the same reasons, Miller [21] also describes the somewhat improved version of the algorithm, diamond-square, as flawed. He then proposes a new — and slightly more complex — version without artifacts that circumvents the problem by sacrificing the requirement that the surface should have to pass through the control points.

A rendering of a terrain generated by the diamond-square algorithm can be seen in Fig. 2.10a.

Another approach to fBm is described in the following subsection.

2.3.2 Implicit procedural techniques

In the previous section, terrains were generated explicitly by evaluating a large batch of noise values at once. When using such explicit techniques, one individual point cannot be generated without generating the rest of the model as well. Implicit techniques, on the other hand, work differently; rather than generating geometry explicitly, they usually rely on a self-contained mathematical model to express the geometry of the terrain. This means that arbitrary data points can be queried independently of the other points. For this reason, it may often be referred to as "point evaluation" [3]. Gamito and Musgrave [22] refers to this as stochastic implicit surface modeling.

In the case of heightmap generation, terrain can modeled as a function

$$z = f(x, y)$$

where z is the height, and x and y are horizontal coordinates. Using this technique, it is trivial to generate heightmaps of arbitrary sizes and resolutions by sampling the function.

A vector field terrain can be modeled similarly:

$$\mathbf{r} = f(x, y)$$

where $\mathbf{r}$ is the displacement vector at the point (x, y).

Implicit methods can also be used to model voxel terrains. Voxel data can be generated by sampling a density function:

$$d = f(x, y, z)$$

where $d > 0$ usually denotes air and $d < 0$ ground (or the other way around). In other words, the terrain is modeled as an isosurface.

In order to model terrain as suggested by Mandelbrot [19] it is necessary to develop an approximation of fBm as a continuous function of two parameters (three for voxel terrain). The rest of this section will explain how such a function can be generated using implicit techniques.

2.3.3 Noise

To construct an approximation of fBm, a common primitive from procedural texture generations, called a procedural noise function, may be used. On its own, procedural noise does not look like fBm, but when different frequencies and amplitudes of the function are added together, it will create an approximation close enough for most uses. A formal definition of procedural noise can be found in Lagae, Lefebvre, Cook, *et al.* [23]. In this project, a procedural noise function is assumed to be an unpredictable continuous function with range approximately $[-1, 1]$. The function will have an approximate frequency of 1 Hz. Ideally, the noise should also be isotropic, meaning that regardless of how you rotate the noise, it will look similar.

Sections 2.3.4 to 2.3.5 will describe different techniques for generating noise.

2.3.4 Value noise

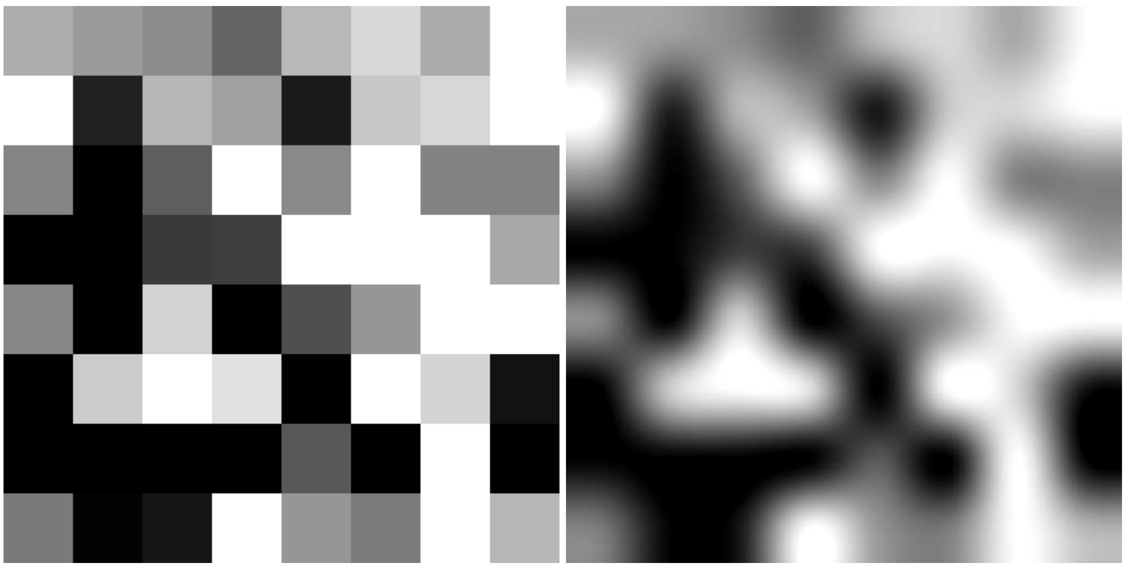


Figure 2.5: Generation of two-dimensional value noise using cubic interpolation. We can clearly see that the noise has artifacts related to the grid direction

One of the most naive approaches to creating procedural noise is to sample white noise in a coarse grid, and then when queried for points find the nearest corners and use an interpolation technique to calculate the value. For most uses, an interpolation technique with at least a second-order continuous derivative should be used. An example of value noise is shown in Fig. 2.5. This technique is rather memory-inefficient and produces noise with directional artifacts (it is not isotropic).

2.3.5 Perlin noise and other types of gradient noise

Gradient noise is a technique slightly similar to, and often confused with, value noise.

At the lattice points, pseudo-random vectors are chosen. When the noise function is sampled, the nearest lattice points are selected. For each selected lattice point, a dot product between the pseudo-random gradient vector and the distance vector to the sample point are calculated. These dot products are then interpolated to calculate the final noise value.

The technique was first described in Perlin [1], along with his famous and widely used implementation of gradient noise, Perlin noise. Perlin won the Academy Award of Motion Picture Arts and Sciences for his contribution in 1997.

Many variations of Perlin noise exist, most of them aim at lowering computational complexity or removing artifacts [23]. Simplex noise, an improvement proposed by Olano, Hart, Heidrich, *et al.* [24], is shown in Fig. 2.6.

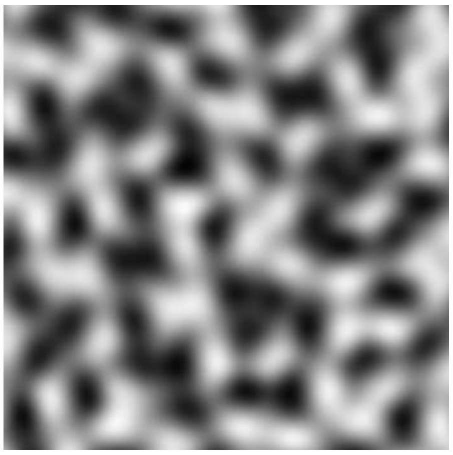


Figure 2.6: A type of gradient noise called simplex noise, an improved version of Perlin noise which eliminates axis-aligned artifacts and reduces computational complexity.

Computing lattice gradients

A naive approach to gradient noise would be to iterate over all lattice points in the domain, and compute and store one unique pseudo-random gradient for each lattice point. The problem with this approach is that it would require huge amounts of memory, especially for high dimension noise functions. The domain of the function would also have to be known in advance. Furthermore, an implementation would suffer a huge performance impact due to cache misses and the cost of creating the lattice values.

In his original implementation of Perlin noise, Perlin [1] solved this by precomputing an array of 256 pseudo-random gradient vectors. Whenever a gradient vector was needed, one of these gradients would then be selected by hashing the lattice coordinates and using the result as an index into the array.

However, Perlin later points out that using 256 pseudo-random gradient vectors for the precomputed array is not necessary and may in fact introduce artifacts because two vectors may be too similar. He argues that using only 12 predefined gradient vectors is sufficient for 3D noise [25].

Several implementations of gradient noise have zero values at the lattice points because of how the points are interpolated.

Perlin [26] suggests an implementation of Perlin noise for GPUs. The implementation uses a slightly different permutation technique than the regular Perlin noise which relies on texture lookups.

Hashing techniques

In the original implementation of Perlin noise, a permutation array was used to select a gradient pseudo-randomly among the 256 random gradients. Such a permutation vector can be constructed by creating an array with the values from 0 to 255 and then shuffling the array.

To use this array to select a gradient, Perlin [1] used the following algorithm:

Listing 2.1: Hashing using a permutation array

```
//permutation array technique, 2D version
int x, y; //lattice coordinates
const int p[256] = [ /* permutation vector */ ];
int p1 = (p[x%256] + y)%256;
vec2 gradient = grad[p[p1]]);
```

In Section 4.4.2 we will present an alternate hashing technique that works well on GPUs.

Interpolating gradient contributions

In Perlin noise, the final noise values are computed by interpolating the individual dot products. This interpolation is done with the help of an interpolation function, $f(x)$, which is 0 when $x = 0$ and 1 when $x = 1$. In his original implementation [1], Perlin used Hermite blending, because of its continuous first-derivative. After the function had been used for a while, it became apparent that in some situations, Hermite blending was insufficient. Specifically when a discontinuous second-derivative caused rendering artifacts. Such artifacts can for example be seen if noise is used for terrains with reflective rendering. To tackle this issue in his improved noise version, Perlin [25] changed to a fifth degree interpolation polynomial. The two polynomials can be seen in Fig. 2.7.

In 1-dimensional Perlin noise, the final value, v, is computed as:

$$t = x - x_0 = x_1 - x \tag{2.1}$$

$$v = v_0(1 - f(t)) + v_0 f(t) \tag{2.2}$$

where x is the point we are querying for, x_0 and x_1 are the nearest lattice points, and v_0 and v_1 are the dot products at each lattice point.

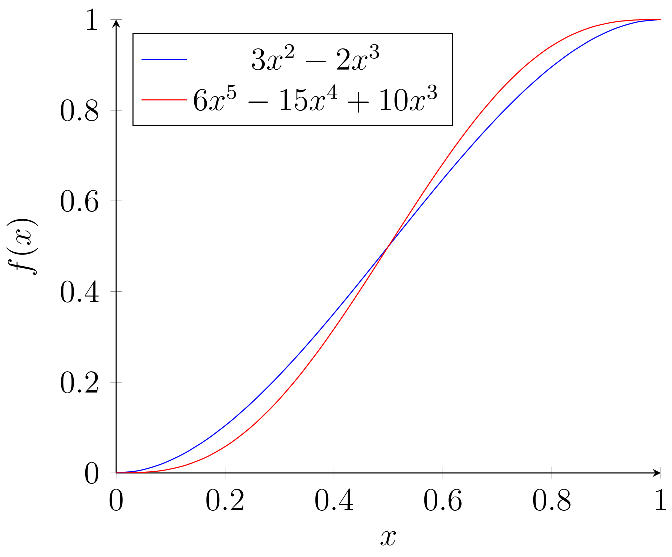


Figure 2.7: Interpolation functions for Perlin noise.

For higher dimensional Perlin noise, the values are simply interpolated for one axis at a time. The results from each interpolation are used as endpoints in the next iteration of interpolations. For this reason, the run-time complexity of Perlin Noise increases exponentaionally with the number of dimensions of the input signal.

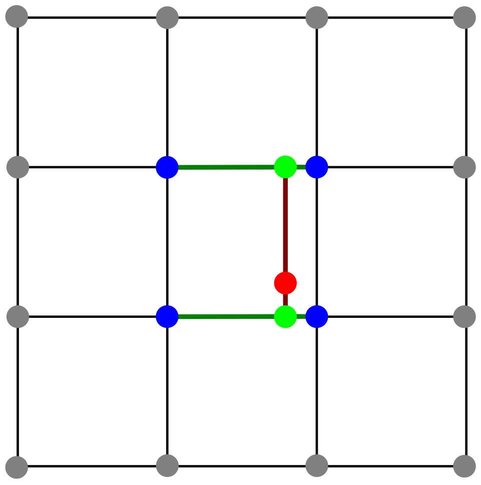
2.3.6 Simplex noise

The in many ways superior algorithm, simplex noise, uses a different approach [24]. Here interpolation is replaced with a summation of independent contributions from each lattice point. In simplex noise, the grid is also formed differently: It is built from a collection of the simplest and most compact shape that can be used to fill space. For two dimensions, this compact shape is a triangle, and for three dimensions, the shape is a skewed tetrahedron. Perlin noise, on the other hand, always uses a hypercube to fill space.

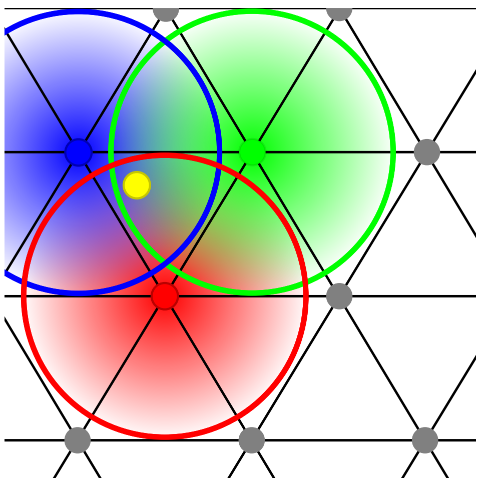
Figure 2.8 shows the grid of two-dimensional simplex noise next to the grid of two-dimensional Perlin noise.

Using the simplest possible shape means that only $N+1$ points are required in the summation of points, where N is the dimensionality of the noise function. This lowers the complexity of the computation from $O(2^N)$ (Perlin noise) to $O(N^2)$.

Although Perlin himself recommends simplex noise as a better alternative [24], Perlin noise continues to be widely used, despite its flaws.



(a) Perlin noise. The first interpolation is along x axis (blue), the second interpolation along the y-axis (green). The final value is shown in red.



(b) Simplex noise. Contributions from three lattice points are summed according to distance to the point (yellow).

Figure 2.8: Gradient 2D-noise construction

2.3.7 Approximating fBm with noise

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.

Fig. 2.9 shows how three layers (called octaves) of simplex noise have been added together to approximate fBm, and Listing 2.2 shows C++-code for the algorithm.

Listing 2.2: Two-dimensional noise-based fBm algorithm in C++

```
float fbm(float x, float y,
          int octaves=3, float lacunarity=2, float gain=0.5) {
    float amplitude=1;
    float frequency=1;
    float sum=0;
    for(int i=0; i<octaves; i++){
        sum += noise(x*frequency, y*frequency)*amplitude;
        amplitude *= gain;
        frequency *= lacunarity;
    }
    return sum;
}
```

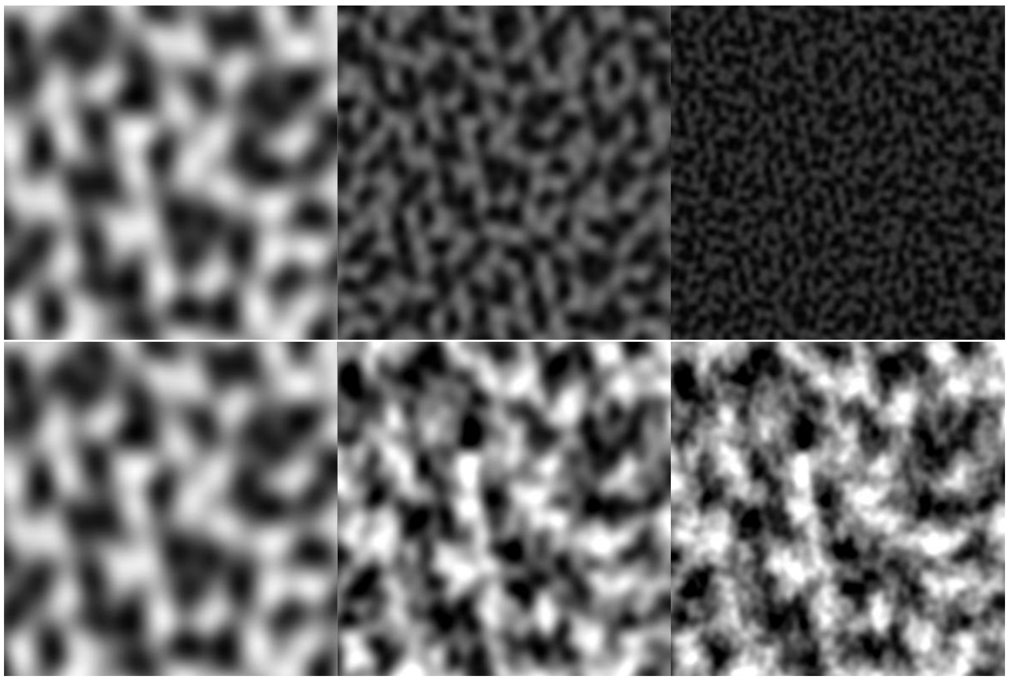


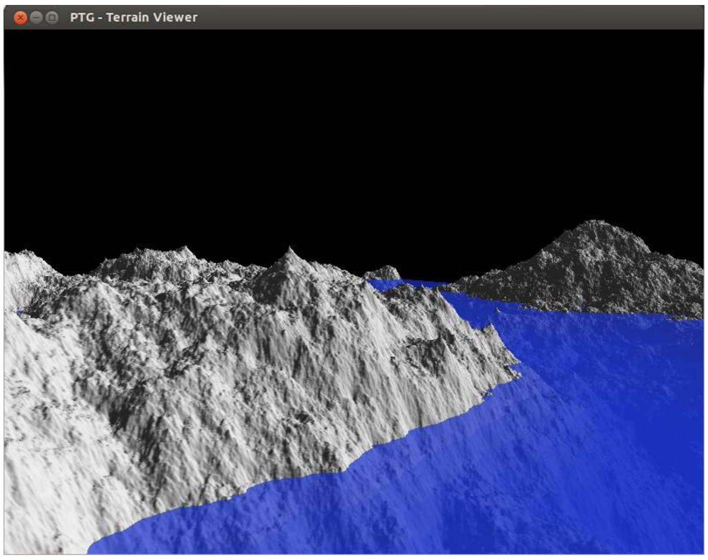
Figure 2.9: Approximating fBm with noise. The top row shows simplex noise with a doubling in frequency and a halving in amplitude for each successive image. The bottom row shows the sum of the images in the top row.

This implementation of fBm is a common choice among applications that need simple procedurally generated terrains. Babington [27] 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 [28] enhanced this implementation by providing a GUI that could be used to adjust the inputs to the fBm function while the simulator was running.

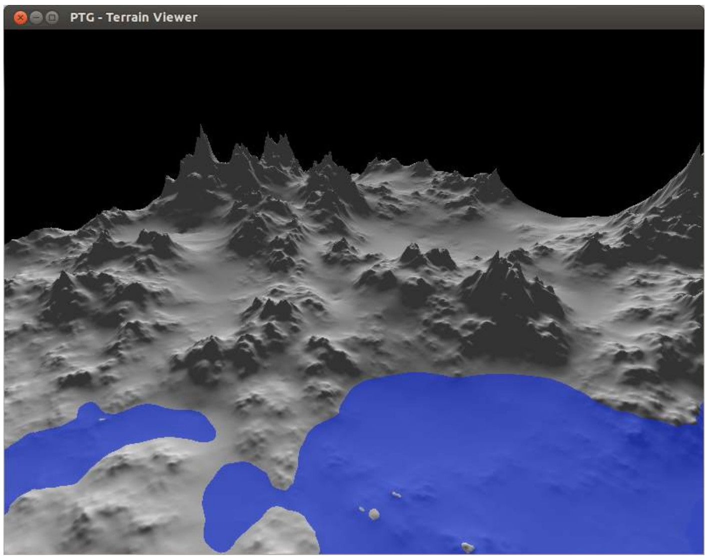
2.3.8 Other uses of noise

As pointed out by Musgrave [3] and Smelik, Tutenel, Bidarra, *et al.* [29], terrains in nature do not behave exactly like fBm. For instance, fBm is too homogeneous and fails to provide local variation of features, such as the roughness of the terrain. This was the inspiration for Musgrave's modifications to the fBm algorithm in Listing 2.2, "hybrid multifractal terrain" (see Fig. 2.10b) and "ridged multifractal terrain".

Musgrave [3] also contains a comprehensive guide on other ways noise can be used as a building block to create a wide range of natural structures. Noise is often used in conjunction with interpolation functions, modulo operations, mul-



(a) fBm terrain using the diamond-square algorithm.



(b) Hybrid multifractal terrain: The gain is adjusted for each octave depending on the value of the previous octave, resulting in a terrain that is rough on mountain tops, but smooth in valleys.

Figure 2.10: fBm-based terrain algorithms rendered by the PTG framework.

tiplication, addition, gamma correction and other operations to get the desired effect. This technique of combining noise in various ways is commonly used by the video game industry for two main purposes: procedural textures and procedural landscapes. Procedural texturing seems to be the most widely adopted application, hence there is a relatively mature set of associated development tools. Also, computation is almost always performed on the GPU. For this reason, procedural shader editors will be discussed in Section 2.8.

As mentioned, these techniques are also used for terrain generation. They can either be used during the development process, in a tool such as World Machine, for offline generation, or for online generation through a general purpose noise library. Such libraries will be discussed in Section 2.5.

A common objection against noise-based heightmap generation in general, is that it lacks user control [29]. This is usually expressed in the context of offline generation where a designer wishes to modify terrain geometry during the generation process. Note that this criticism becomes irrelevant in the case of online generation, where the goal is to design a completely autonomous generation process. In fact, Smelik, Tutenel, Bidarra, *et al.* [29] also gives praise to noise-based algorithms for producing "natural, mountain-like structures".

2.4 Simulating erosion

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.

Thermal erosion is the result of temperature changes breaking up rocks and transporting sediments downwards where the incline is steep. Thermally eroded terrains typically have grooves and V-shaped valleys. The phenomenon was first simulated by Musgrave, Kolb, and Mace [2] in order to improve the quality of terrains generated with fBm.

Musgrave, Kolb, and Mace [2] simulated thermal erosion by iteratively modifying heightmap terrains. In each iteration, the altitude of each cell is compared against the altitude of its neighbors. If the height difference is bigger than some talus angle, T, a percentage of the height difference, c_t, is transferred to the current altitude

$$a^u_{t+1} = \left\{ \begin{array}{ll} a^u_t + c_t(a^v_t - a^u_t - T) & \text{if } a^v_t - a^u_t > T \\ a^u_t & \text{if } a^v_t - a^u_t \leq T \end{array} \right\} \tag{2.3}$$

where a^u_t is the altitude of point u at time step t and v is a neighbor of u. Fig. 2.11 shows the terrain from Fig. 2.10b after running a thermal erosion simulation.

This model of thermal erosion was later used by Benes and Forsbach [15] on layered heightmap terrains in order to simulate erosion on more complex terrains consisting of layers of sediments with different attributes. By assigning a custom talus angle to each layer, it is possible to simulate erosion on hard materials, such as rock, and soft materials, such as sand, in the same model.

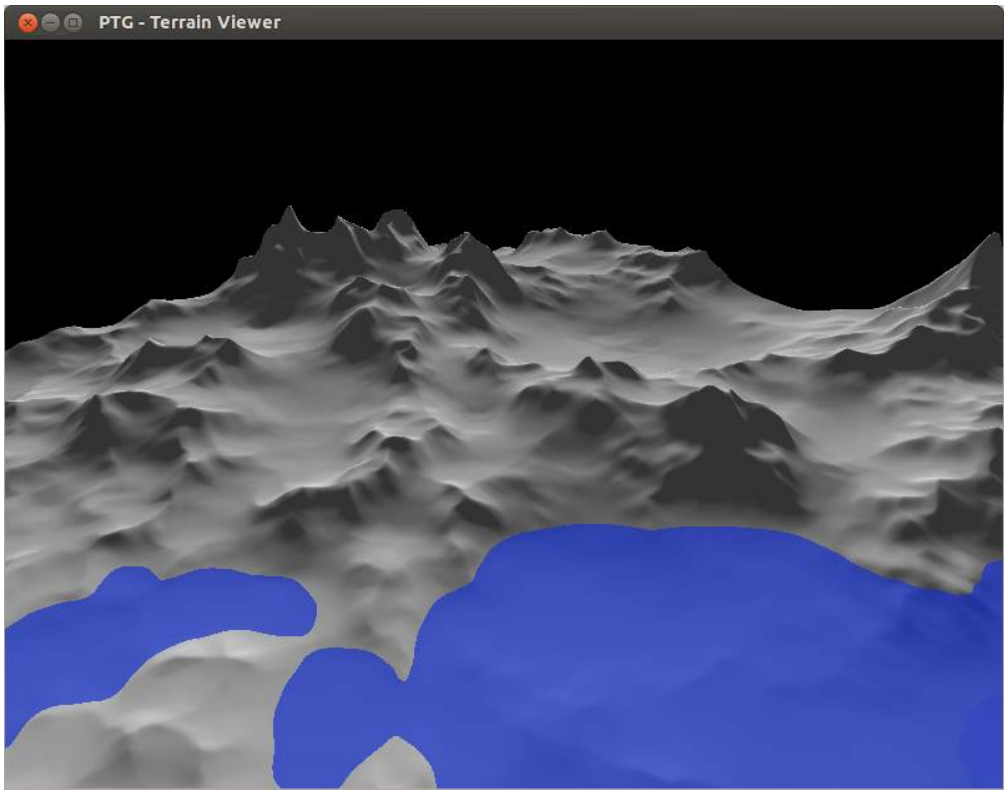


Figure 2.11: Running thermal erosion on a hybrid multifractal terrain.

Olsen [30] implemented thermal weathering with an emphasis on terrain generation for real-time strategy games. Olsen [30] used a slightly modified version of Musgrave, Kolb, and Mace [2] and implemented it on the GPU. Olsen [30] were able to erode a 1024×1024 heightmap in less than 4 seconds with hardware from 2004.

Another type of erosion simulation was also presented by Musgrave, Kolb, and Mace [2], namely hydraulic erosion. This simulation tries to capture the effect on water transporting sediment when flowing downhill. Like thermal weathering, the simulation iteratively modifies a heightmap. Two separate maps are maintained to keep track of water and sediment suspended in the water. In each iteration, water is transfered downhill in a manner according to the slope at that position. If the water is flowing rapidly, sediment is suspended in the water. If water flows slowly, sediment is deposited.

Olsen [30] also tried to implement this algorithm on the GPU, but found it to be too slow to be used in video games. Since their attempt in 2004, the performance of GPUs have increased rapidly, and Mei, Decaudin, and Hu [31] managed to implement the algorithm with interactive performance.

A different approach to hydraulic erosion with interactive results has been developed by Krištof, Beneš, Křivánek, *et al.* [32].

2.5 Noise libraries

As mentioned in Section 2.3, designing procedural terrains may be done by programming against a general purpose noise library. In Section 2.5.1, a mature and widely used noise library, libnoise, will be described. In Section 2.5.2, a less common, but more interesting approach, ANL, will be discussed.

2.5.1 libnoise

libnoise is an open-source (LGPL) C++ library for noise generation. The library contains implementations of many popular algorithms, such as the ridged multi-fractal and Perlin noise.

The library has a concept called a "noise module", which is a class interface that contains a method, "`double GetValue(double x, double y, double z)`". An implementation of the interface may contain references to other modules which may be responsible for computing parts of the result. These other modules are referred to as "source modules". Since there are modules available for most common mathematical operations, it is possible to utilize this library to synthesize noise in the way described by Ebert, Musgrave, Peachey, *et al.* [33].

libnoise is widely used, and perhaps the most popular noise library available. Recently, it was used by Davis [34] to create a voxel terrain generator.

The library is also used for many GUI applications for offline texture and heightmap generation. TerraNoise is one such application, aimed at producing heightmap textures [35]. Noise Mixer is another similar application. Although these applications are aimed at creating heightmaps, they lack 3D previews of the terrain, showing only the heightmap texture.

2.5.2 Accidental Noise Library

Accidental Noise Library (ANL) is another noise library for compositing noise functions. Like libnoise it allows the construction of a noise-based function through a similar concept, also called "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. Examples of how this scripting language is used may be found in Listing C.3 and Listing C.4.

In Tippetts [36], the lead developer of ANL gives an introduction to the LUA API of the application, as well as an explanation of how it can be used to generate a rich voxel terrain including cave networks.

The library is licensed under an open-source license, MIT. It runs on the CPU only and does not take advantage of the computing power of GPUs.

2.6 Offline terrain generators with procedural features

Numerous tools exist for generating and exporting heightmaps and meshes during development time. These tools are typically used by designers to create static worlds for traditional games. New terrain data can not be generated by games during run-time using these tools.

Examples of such tools are World Machine, Lithosphere, Terragen and Bryce.

2.6.1 World Machine

World Machine is a commercial terrain creation tool [5]. It uses a network-based (graph) user interface to combine input and output of algorithms and filters, as well as with user-defined data. It features a low-resolution preview of the terrain that is updated in real time. A screenshot of the tool can be seen in Fig. 2.12.

Complicated hierarchies of algorithms can be combined into "macros" to create simpler and reusable abstractions.

Generated terrains can be viewed in real-time through a 3D-view or as a texture.

There is no source code available for the tool, so it is not possible to refactor the tool into a library in order to generate terrains online.

This tool is widely used in the development process of commercial games [10], [12].

2.6.2 Lithosphere

Lithosphere is an open-source terrain generator developed by Florian Bösch. It lets you design a terrain by editing a flow graph of algorithms and functions while viewing the resulting terrain in real-time. Although the terrain preview is real-time, there is no functionality for generating the terrain inside other applications (as a library). Terrains can only be exported as heightmaps or meshes.

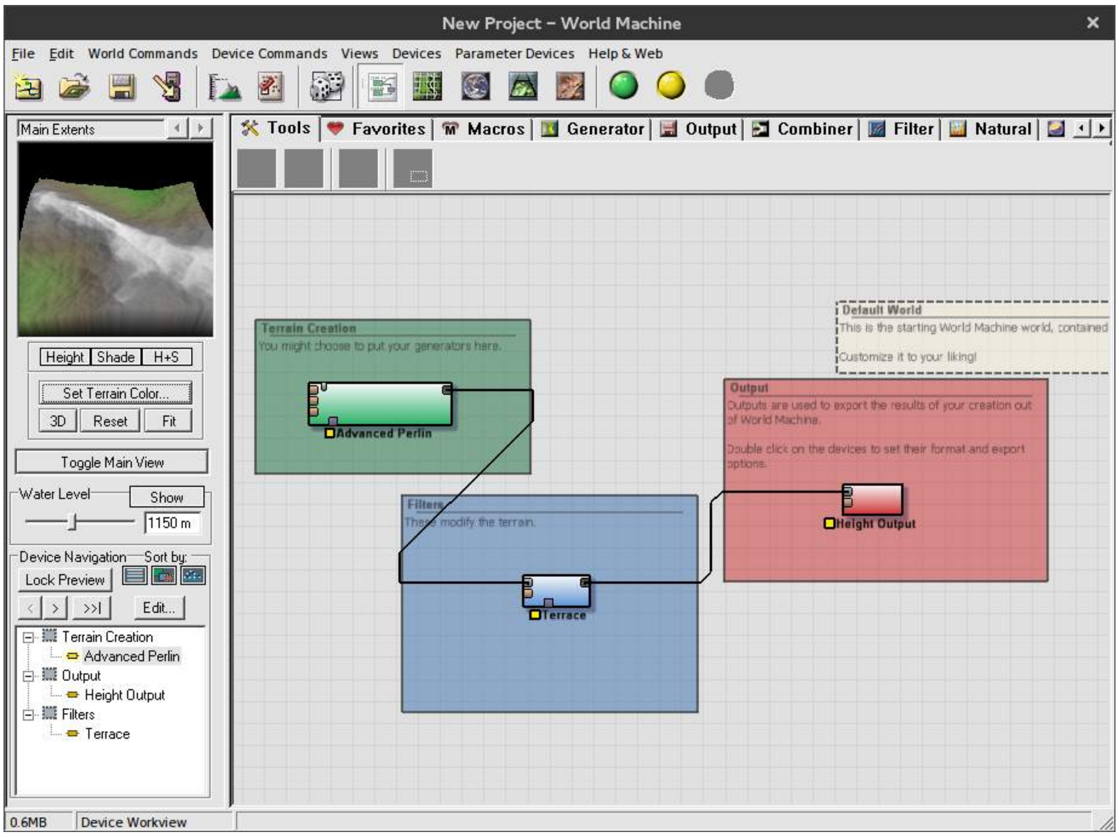

New Project – World Machine
File Edit World Commands Device Commands Views Devices Parameter Devices Help & Web
Main Extents
Height Shade H+S
Set Terrain Color...
3D Reset Fit
Toggle Main View
Water Level Show
1150 m
Device Navigation Sort by:
Lock Preview
Edit...
Terrain Creation
Advanced Perlin
Output
Height Output
Filters
Terrace
0.6MB Device Workview
Tools Favorites Macros Generator Output Combiner Filter Natural
Terrain Creation
You might choose to put your generators here.
Advanced Perlin
Filters
These modify the terrain.
Terrace
Default World
This is the starting World Machine world, contained
Customize it to your liking!
Output
Outputs are used to export the results of your creation out of World Machine.
Double click on the devices to set their format and export options.
Height Output

Figure 2.12: A screenshot of World Machine. A preview of the terrain is shown in the top left, and a flow-graph editor is shown in the middle.

Although the program is open-source and it may be possible to extend it in order to generate terrains online as a library, it is currently released under AGPL, which makes it incompatible with many applications, including closed-source games [37].

2.7 GeoGen

The bachelor thesis of Matěj Zábský, GeoGen [38], is an open-source procedural heightmap generator that can be used for real-time generation. It consists of a "studio" where you can write terrain scripts and preview them, and a library that can read those scripts and generate geometry online.

While the approach shortens the feedback loop for terrain editing by offering a convenient way to generate previews, the editor still assumes a certain proficiency with scripting languages, and the user has to familiarize himself with the Squirrel scripting language and the API of GeoGen.

GeoGen also depends heavily on explicit algorithms, which causes a number of issues to arise, the most important being that there is no easy way to obtain a subsection of the world without generating the entire heightmap. Consequently, the entire virtual world has to be able to fit in a single heightmap. This puts a limit to the size and detail of the world, thereby removing many benefits of procedural generation.

Another issue with explicit algorithms is that they are often difficult to parallelize. In fact, all computation in GeoGen take place on the CPU, causing an excessive amount of time to be required for some operations. The time passing between a modification is made, until the preview is updated, is usually in the order of seconds or minutes. For example, results from Zábský [38] show that generating an eroded 2048×2048 heightmap required over 7 minutes. In comparison, the GPU implementation of Olsen [30] required only 4 seconds on 7 years older hardware.

To generate an uneroded heightmap the maximum size supported by the Unreal Engine, 8192×8192, GeoGen generally required 1 to 3 minutes depending on the terrain model [38]. So while the previewing of terrains have been streamlined, it is certainly not possible to view changes to a high resolution map in real time.

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

2.8 Procedural shader editors

Procedural shader editors is a category of development tools that are usually used to design procedural textures, often called materials. While not commonly applied for generating terrains, procedural shader editors have a number of qualities required by a design tool for online procedural terrain generation:

1. Many of the same algorithms are used. Noise synthesis is a common technique for both terrains and textures.

2. Execution on the GPU. Terrain generation computation is not required to take place on the GPU, but would significantly improve performance.

3. A Real-time preview is essential for an efficient work-flow, and most importantly:

4. The model created has to be expandable during run-time of the game.

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 that of World Machine. Such shader editors are usually engine-specific [9], [39], [40]. There are also a few shader editors available, which create shaders usable by multiple game engines, such as Allegorithmic Substance Designer [41]. Also, Microsoft recently obtained a patent for a “visual shader designer” [42].

2.9 OpenGL

OpenGL is a platform-independent graphics rendering API [43]. It is an interface that makes it possible to write portable rendering code for a wide range of hardware and software. The specification for OpenGL is language agnostic, but implementations of the API are typically written in C. Bindings for many other languages are available as well, either with official support, such as WebGL, or through third party wrappers around a C implementation, such as GoGL.

The interface is composed of numerous functions which may be called to influence a state machine.

In Section 2.9.1, a quick overview of the OpenGL pipeline is given. In Sections 2.9.2 to 2.9.6, the different programmable shaders are briefly discussed, as well as their intended usage. Section 2.9.7 contains an overview of the OpenGL shading language.

Since our GUI application targeted OpenGL 3.0 the geometry, tessellation, and compute shaders will only be discussed briefly. This decision is addressed in Section 4.5.

OpenGL is managed by a non-profit organization, called the Khronos Group.

2.9.1 The rendering pipeline

When OpenGL is used, a client program typically runs on the CPU. The client program issues calls to OpenGL, then the OpenGL implementation of the client system is responsible for handling the rendering hardware of the platform. This includes transferring textures to GPU memory, compiling shader code and making sure the program is executed by the GPU.

With the release of OpenGL version 3.0, OpenGL shifted away from a “fixed pipeline” towards a “programmable pipeline”, in the sense that it leaves more control of the graphics hardware to be manipulated by developers. This is done by introducing programmable shaders and deprecating functionality that assumes a specific implementation of those shaders. In this section, only OpenGL 3.0+ will be described, as the new API is simpler, smaller and much more powerful than previous versions.

2.9.2 Vertex shader

The vertex shader is traditionally used to decide the final screen position of a geometric point or vertex. It transforms a single incoming vertex into a single outgoing vertex. The main task of the vertex shader is usually to multiply the raw vertex from the geometric model with a model-view-projection matrix in order to get the screen position.

The vertex shader passes on or computes information needed by the fragment shader. Such information may include vertex normals and texture coordinates, or it may simply be a single color value if a simple shading model, such as Gouraud or flat shading, is used.

2.9.3 Fragment shader

The fragment shader is traditionally used to decide the final color and depth value of a specific fragment or pixel. The inputs of the fragment shader can simply pass on a color value computed in the vertex shader (if using flat or Gouraud shading), or do more advanced calculations such as adding contributions from different light sources taking into account a supplied normal, if using a more advanced shading model, such as Phong shading.

2.9.4 Geometry shader

A geometry shader is an optional shader introduced in OpenGL 3.2. The input of a geometry shader, is a single renderable primitive, such as a line or a triangle. The output is zero or more primitives.

2.9.5 Tessellation shaders

Tessellation shaders were introduced in OpenGL 4.0. Their task is to subdivide a patch of vertex data into smaller primitives. Tessellation shaders can be very useful for rendering terrains because they make it possible to draw high-resolution primitives only where they are needed.

Although they are quite useful, tessellation shaders were not used in this project, since they require OpenGL 4.0+, which is not yet supported by open-source Linux drivers. Tessellation shaders are nevertheless relevant because game engines taking advantage of our framework may want to use them.

2.9.6 Compute shaders

Compute shaders were introduced in OpenGL version 4.3. They specialize in computation of values, and would have been well suited for batch computation of terrain data. Compute shaders allow a programming model more similar to CUDA and OpenCL. They would have been a great alternative for terrain generation, unfortunately they are not supported well enough in consumer hardware and software to be used in this project.

2.9.7 The OpenGL shading language

The OpenGL shading language, or GLSL, is a high-level programming language for writing shader programs that can be executed by a GPU.

Shaders are parts of a shader program that may run on the GPU.

2.9.8 Noise generation on the GPU

When Perlin wrote his original noise function in 1983, GPUs did not exist yet. Consequently, Perlin noise and many of its most common derivatives are optimized for CPU rendering. Here, we will look at different methods for synthesizing noise on the GPU using OpenGL.

Perlin suggested an implementation for use in pixel shaders [26]. He made various modifications to the improved version he proposed in Perlin [25], such as using a different hashing technique, which relies on textures. He also sacrificed the fifth order interpolation polynomial in order to take advantage of hardware interpolation.

Green [44] proposed a different GPU implementation in the next issue of GPU Gems. This implementation does not make the same algorithmic trade-offs as Perlin [26], and in fact gives identical results to the CPU version in [25]. Green [44] stores the permutation vector in a 2D texture of height 1.
A version which did not use any texture look-ups was later proposed by McEwan, Sheets, Richardson, *et al.* [45]. The approach is entirely computational, and uses a slightly different hashing technique called a permutation polynomial, instead of Perlin's permutation array. A permutation polynomial is a function that can be used to permute a sequence of integers similarly to a lookup in a permutation array. $(6x^2 + x) \, mod \, 9$ is an example of such a function, since $(0, 1, 2, 3, 4, 5, 6, 7, 8) \mapsto (0, 7, 8, 3, 1, 2, 6, 4, 5)$ [45].

In McEwan, Sheets, Richardson, *et al.* [45], the selection of gradients is also slightly different; instead of storing precomputed gradients, gradients are selected from the surface of a cross polytope surface. In 2012 the implementation was twice as slow as a version that used texture lookups. However, according to McEwan, Sheets, Richardson, *et al.* [45] this gap in performance is not as large as it seems, since the GPU is often assigned other work as well and texture bandwidth tends to be a scarce resource, hence there is often an excess of unused computation power. The authors tested their implementation for both Perlin noise and simplex noise in 2, 3 and 4 dimensions.

2.10 Terrain rendering

A terrain heightmap is usually represented by a regularly spaced grid in the form of a texture. In order to draw a three-dimensional picture of the terrain, this representation has to be translated into triangles for the GPU to rasterize.

A naive approach would be to create a vertex for each texel, with coordinates

$$(x_i S_w, y_i S_w, h_{x_i,y_i} S_h)$$

where h_{x_i,y_i} is the height value at the texel index (x_i, y_i). S_w and S_h are constants that scale the vertices in width and height. After doing this, two triangles could be formed for each set of four adjacent vertices.

This will draw a uniform terrain with an equal amount of detail everywhere using all the available terrain information. There are several problems with this approach, however:

- Terrains are usually rendered using a perspective projection. This means that areas of a terrain that are far away from the observer will appear smaller than those close to the observer. Hence, much performance is wasted on drawing tiny triangles that may even be smaller than one pixel.
- Terrain roughness may vary. Spending an equal amount of detail on flat plains and rocky mountains does not make sense.
- Some areas may not be visible from the observer's orientation. Generating and making calls to draw these areas wastes resources.

In order to draw huge terrains that still have an acceptable level of detail close to the observer, a level-of-detail or LOD algorithm can be applied. Section 2.10.1 presents several such algorithms.

2.10.1 Level-of-detail algorithms

ROAM

ROAM, or Real-time Optimally Adapting meshes is a level-of-detail algorithm [46]. A terrain is represented as a binary tree of isosceles right triangles. A node in the binary tree may either be a leaf node representing a single triangle to be drawn on screen, or an internal node representing a collection of triangles that together form a larger triangle.

Leaf nodes may be split in order to provide a higher level of detail, or merged to reduce the triangle count. If triangles are split, siblings or parents may also be split in order to prevent T-junctions. The technique was developed in 1997 for CPUs, and GPUs have become much more powerful since then. The algorithm is difficult to adapt for GPUs and is therefore not much used today.

2.10.2 Geometry Clipmaps

Geometry clipmaps is another LOD-algorithm described by Losasso and Hoppe [47] and patented by Microsoft. The technique stores vertex buffers in GPU memory that are generated while the viewpoint moves. Rectangular grids of a fixed resolution are drawn centered around the viewpoint. Each grid is twice the size of the next LOD-level. An example of clipmaps can be seen in Fig. 2.13. The vertex buffers are updated incrementally by the CPU.

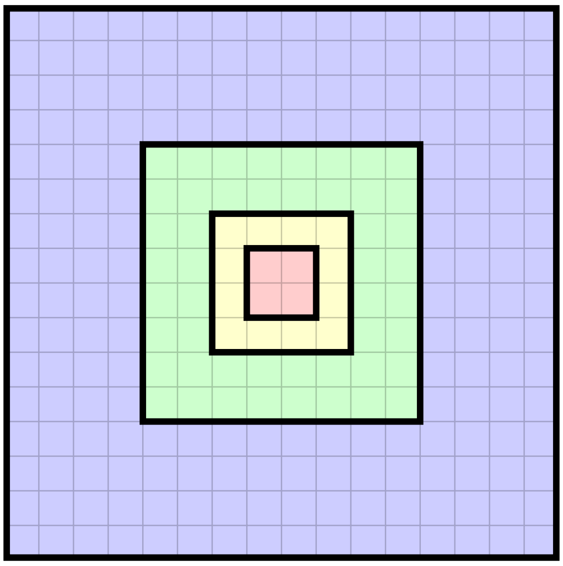


Figure 2.13: Geometry clipmaps as seen from above. Each color represents a LOD-level. All LOD-levels have the same number of vertices in width and length.

Asirvatham and Hoppe [48] improved on this implementation by moving more of the computation to the GPU. This implementation also allows terrain synthesis to be performed by the GPU, making it highly compatible with GPU-based procedural terrain generation.

The algorithm is used in state-of-the-art game engines [12].

2.10.3 Continuous distance-dependent level of detail

Continuous distance-dependent level of detail, or CDLOD, is a recent LOD-algorithm described by Strugar [49]. It works by dividing terrain geometry into a quadtree of square terrain patches, similarly to how ROAM divides a terrain into a binary tree of triangles.

Each leaf node represents a fixed resolution patch of terrain with a width usually between 64 and 256. Its biggest advantage over clipmaps is that CDLOD selection takes into account the height of the observer. This means that the algorithm can adapt to large changes in altitude during run-time. An illustration of the quad-tree structure of CDLOD can be seen in Fig. 2.14.

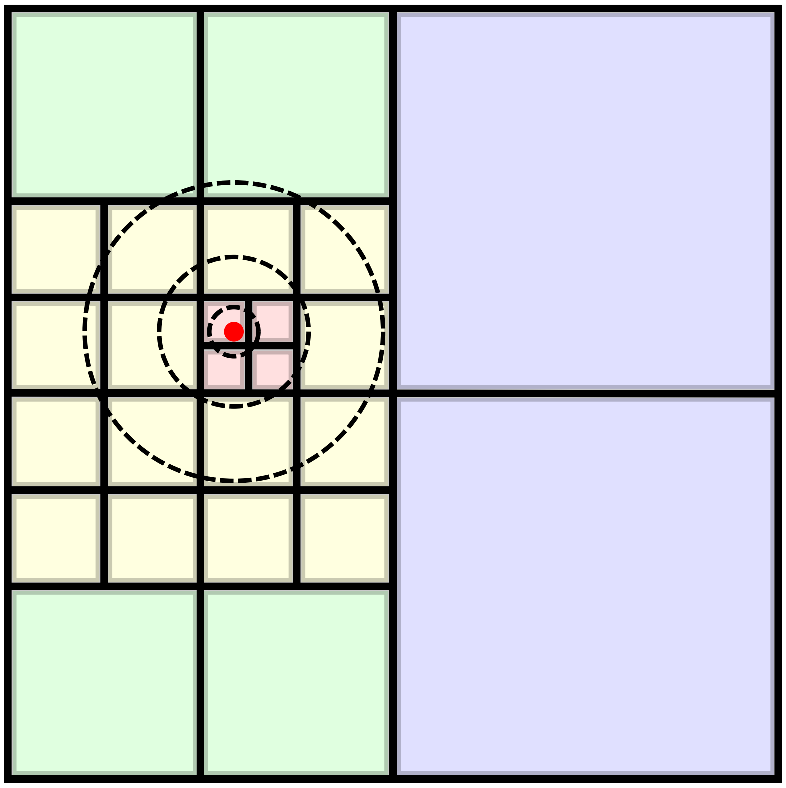


Figure 2.14: The CDLOD algorithm. Each square represents a heightmap of width 128. The circles indicate the distances at which each LOD-level is activated.

If a tessellation shader is available, a patch may be rendered by passing the four corners of a terrain patch to the tessellation shader. If a tessellation shader is not available, it is possible to tessellate a flat grid using the CPU. One tessellated grid may be used to render all patches, because height data can be kept separate from the grid vertices or generated using procedural techniques.

Transitions between LOD-levels are also seamless, since four vertices will smoothly morph into one before the LOD-level is changed, thus preventing a popping effect often seen with many other techniques. This transition is applied proportionally to the three-dimensional distance to the observer.

A variant of CDLOD has been used by Babington [27] to improve the terrain rendering in the NTNU HPC-lab snow simulator.

2.11 Qt

Qt is an open-source framework for developing cross-platform GUI applications. It is currently developed by the Finnish software company Digia.

QtQuick is a recent addition to the Qt framework, allowing user interfaces to be designed declaratively and separate from application logic, using a markup language called QML, in conjunction with JavaScript [50].

QtQuick can interface with a C++ program if it is programmed using the Qt framework. Objects derived from QObject can be accessed through QML and JavaScript through the QtQuick engine.

Qt uses a system of signals, slots and properties, in order to notify other parts of the program when data has changed.

A slot is simply a public function of a class, while a signal corresponds to an Observable in the Observer pattern [51]. Slots may subscribe to signals, so that they are called when the signal is “emitted”.

Chapter 3

Method

In this chapter, we describe our approaches to answering the research questions in Section 1.4. This starts with Section 3.1, which is an explanation of how our definition of the terrain generation problem differs from that of other research and commercial tools.

In Section 3.2, we propose an answer to RQ1. We present a representation for functional composition of stochastic implicit terrains that can be implemented deterministically on a wide variety of platforms. In order to verify this claim, and also to give answers to RQ2, a proof-of-concept terrain-editing framework has been developed.

While the main goal of the framework is to help answer the research questions, it is also intended to be useful as middleware for the game development industry. The resulting requirements for the framework are described in detail in Section 3.3. These requirements are then used to guide the development of the framework architecture, which is outlined in Section 3.4.

Section 3.5 explains our approach to satisfying RQ2 using parallel computation on the GPU, and also gives a rationale for why we chose OpenGL and GLSL as our computation platform.

Finally, in Section 3.7, we will explain the measures taken to assure the validity of our answers to the research questions.

3.1 Novelty of approach

Most current tools for procedural terrain generation focus on how to augment the development process of traditional games using procedural techniques [5], [7], [35], [37], [52]. In this project, however, the aim was to assist in the development of an emerging genre of games featuring content generated procedurally during run-time. Examples of such games are: *Minecraft*, *Terraria*, *Spore*, and *Elite: Dangerous*.

In these games, procedural generation is not merely a way to reduce development costs, compress data, or provide more detail, but a main feature and selling point of the game as well. They benefit from two of the effects of procedural generation often ignored by procedural modeling tools, namely increased replayability and the increased vastness of the virtual worlds. Many of these games feature a unique world during each game session, and a world so vast it could not possibly have been designed by hand, or even observed in its entirety by a designer.

One of the main arguments of recent works [29], [53], is that procedural modeling tools should integrate with the kind of manual editing operations designers are used to, preferably letting the designer iteratively edit or refine the terrain after it has been generated procedurally. This feature, however, is impossible to implement for the subset of games described above, since the terrain has to be generated during runtime after the game has been shipped.

When procedural techniques are applied to empower replayability and vastness rather than shortening development time and compressing data in a traditional static game, focus is shifted away from editing *one individual* terrain towards ways to control and edit *types* of terrain.

Few existing tools are capable of visualizing terrains that can later be generated online when the game is running. The only tool capable of this known to the author, is GeoGen, the bachelor thesis of Matěj Zábský [38]. As discussed in Section 2.7, there are several flaws in the implementation of GeoGen. We have addressed these issues by using only implicit procedural techniques rather than mixing explicit and implicit techniques.

Because no editors for online procedural terrain modeling are available, tools are often developed from scratch for each game engine. For many projects, this means that terrain modeling is often done exclusively through editing code.

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.

No other tool known to the author fulfills all the following requirements (see Section 1.3):

- Online generation through a library
- A real-time 3D preview
- A flow-graph-based interface

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.

3.2 Concepts

This section describes the most important concepts of our approach, namely how a terrain generator may be expressed as a graph of function calls.

3.2.1 Terrain representation

Implicit procedural surface modeling, as described in Section 2.3.2 and Section 2.5, are used 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 [4] refers to this as a hypertexture hierarchy.

A node's inputs can be specified as constants, or by connecting an edge from an input to another node's outputs. An output of a node can be connected to any number of inputs, but an input can only be connected to one or zero outputs.

There can be no cycles in the graph, as that would make a node depend on the result of its own computation.

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.

3.2.2 Modules

A node in the function graph will be referred to as a “module”. Several names were considered: World Machine has a similar concept, named “devices” [5], while Lithosphere simply calls them “nodes”. ANL has “modules” and libnoise has a more specific notion of “noise modules” [54], [55]. Although “module” is a relatively vague term, it seems to be the most widely adopted.

A module will have a number of inputs and outputs according to its “module type” (see Section 3.2.3). The inputs may be assigned a constant value, or they may be assigned to the output of another module in the graph.

3.2.3 Module types

A “module type” may be considered as the blueprint for a module. When modules are created, they will always have a corresponding module type. The module type describes what kind of operation the module represents, i.e. which inputs and outputs are available, and the specifics of how those outputs are computed. They also specify the default values for a module’s unlinked inputs.

Two examples of module types are “abs” and “fbm2”. “abs” has a single input, and a single output. The operation consists of taking the absolute value of the input and assigning it to the output. “fbm2” is a more advanced module type, which represents the algorithm for two-dimensional fractional Brownian motion. It has a single output, and many inputs.

A function definition in a functional programming language, such as Lisp, is a close approximation of a module type. In fact, our approach can be used to generate function definitions from module types. Taking this analogy further, a module may simply be thought of as a function call.

A list of available module types can be found in Appendix A.6.

3.2.4 User types

In addition to the built-in module types described in Appendix A.6, it is also possible to define custom module types; these are referred to as “user types”. A user type consists of a module graph with two special input and output modules. The input module has one output corresponding to each of the user type’s inputs, while the output module has one input corresponding to each of the user type’s outputs.

User types make it possible to break up large and complex graphs into smaller sub-graphs by encapsulating the sub-graph in a new module type. This has two advantages, abstraction and reuse. Some sub-graphs may be very complicated and hard to manage. By encapsulating a complicated graph it is possible to give meaningful labels to inputs and outputs. Sometimes, a user may notice that a certain graph structure will occur frequently in different graphs, or in different places in the same graph. By encapsulating such a structure, it can be reused in different places, reducing the complexity of the graph, and providing non-destructive editing for the encapsulated structure.

An example of this could be that a sub-graph used for calculating terrain heights for mountain-like terrains can be encapsulated in a mountain user type. This user type could subsequently be used in other graphs that might, for instance, blend mountains, plains and coastal landscape into a more varied world.

Composition of user types is also explained in the figures in Appendix B.

3.2.5 Signal types

In our approach, inputs and outputs of modules use a very simple type system. The only feature distinguishing signal types, is their dimensionality. All types are simply vectors of floating point numbers. By providing a dimensionality for signals, it becomes easy to encapsulate a position as a single signal, instead of separate x, y, and z signals.

3.2.6 Metadata

Modules and module types have two additional attributes; names and descriptions. Names are mandatory, because they are used as identifiers, while descriptions are optional. Descriptions may simply be regarded as code comments or documentation.

3.2.7 Formal model

A module graph can be modeled as a system of equations. For each module, m, **except** the input module, m_{in}, we have an equation

$$\mathbf{O_m} = f_m(\mathbf{I_m}) \tag{3.1}$$

where $\mathbf{O_m}$ is the vector of output values for the module, $\mathbf{I_m}$ is the vector of inputs to the module, and f_m is the function represented by the module type.

We also have a set of equations

$$I_{m,i} = O_{n,j} \tag{3.2}$$

representing input i of module m being connected to output j of module n. Finally, we have the set of equations for inputs not linked to outputs

$$I_{m,i} = C_{m,i} \tag{3.3}$$

where $C_{m,i}$ is a known constant.

When this system of equations is used to query for values, we augment the set of equations with a final set of equations for the outputs of the input module, m_{in}, which we omitted earlier:

$$O_{m_{in},i} = C_{m_{in},i} \tag{3.4}$$

When this final set of equations is added to the system, it becomes possible to solve for any output or input. In most situations, there will be a specific output module, m_{out}, and we will solve for $\mathbf{I_{m_{out}}}$.

3.2.8 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.

In libnoise, functions (modules) are assumed to take three inputs and return a single output. For applications that only need two-dimensional functions, such as for heightmap generation, the third input is simply ignored and set to zero. This behavior causes the more expensive three-dimensional noise functions to be invoked regardless of whether they are needed or not, making performance suffer.

In ANL, this problem is solved by requiring all modules to implement all supported signatures. This means that every module in ANL has five different implementations (one for each supported dimensionality). This allows higher dimensional queries to be made, as well as for optimizations when making queries in lower dimensions. Although this approach may seem more dynamic, it is really just five separate implementations of the same approach as libnoise. The function graph is still homogeneous, meaning that all function calls used to answer a query will have the same signature as the first call.

When one module has another module as its source in libnoise or ANL, it means that the source module will be queried with the arguments specified by the calling module; usually they are identical to the arguments in the call to the first module. In our approach, there is no concept of a source module, instead there is the concept of inputs being connected to outputs.

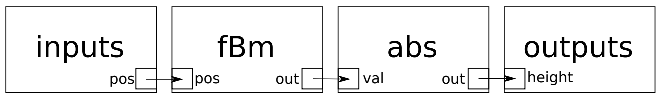
While edges in an ANL or libnoise graph correspond to function calls, edges in our approach correspond to outputs being assigned to inputs. In our approach, the nodes, or modules, themselves are the function calls, while modules in ANL represent callable functions.

The difference is illustrated in Fig. 3.1, where the example portrays a simple fBm heightmap terrain filtered through a module that computes the absolute value using both representations.

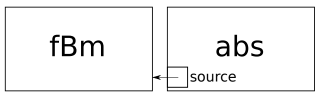
Using our approach, the fBm module has an explicit position input, while in the ANL representation, this position is provided in the function call itself.

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



(a) Our approach. Values flow from left to right.



(b) Approach of ANL, libnoise. Function calls flow from right to left.

Figure 3.1: A function calculating the absolute value of fBm represented by two different graph metaphors.

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

In this section, the research questions from Section 1.4 will be revisited and a set of functional and non-functional requirements will be developed to make sure the developed software works as intended.

3.3.1 Real-time performance

In order to assess the outcome of an editing operation, the user needs to receive visual feedback in the form of a terrain preview. Many procedural terrain generation tools require an explicit user action to produce such a preview [5], [38], [56], while a few select tools will automatically update the preview on every change [37].

Victor [57] makes a good argument for why it is important to have visible results immediately available:

> Creators need an immediate connection to what they create. And what I mean by that is: When you are making something, if you make a change, or you make a decision, you need to see the effect of that immediately. There can't be a delay, and there can't be anything hidden. Creators have to be able to see what they're doing. [. . .] To be able to try ideas as you think of them. If there is any delay in

> that feedback loop, between thinking of something and seeing it, and building on it, then there is this whole world of ideas which will just never be. These are thoughts that we can't think.

He argues that for a creative process, the amount of time passing between an editing operation and the moment its observable results are available is absolutely crucial. If the delay is too long, the creator will tend to experiment less, and will have less control over what she is creating. For this reason, we require the preview to be updated instantaneously as the user adjusts values. This is also our motivation behind formulating research question RQ2.

Just how fast is "instantaneous", and what is the maximum delay between input and display that can be tolerated? This is an important question to answer in order to set a specific requirement for just how fast an implementation needs to be in order to satisfy RQ2. Swink [58] gives the following three categories for continuous real-time controls:

Below 50 ms response feels instantaneous.

100 ms delay is noticeable but ignorable.

Above 200 ms response feels sluggish.

Ideally, the delay would be below 50 ms, but a delay of 100-200 ms is also acceptable.

In order to support such a short delay, terrain data needs to be generated efficiently. 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. In the following is given 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:

$$v = lw^2 - (l-1)(\frac{w-1}{4} + 1)^2 = lw^2 - \frac{1}{16}(l-1)(w+3)^2 \tag{3.5}$$

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:

$$v_{visible} \approx \frac{\alpha}{2\pi} v \tag{3.6}$$

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* [10] has been inserted into Eq. (3.5) and Eq. (3.6). Inserting $l = 5$, $w = 1025$, and $\alpha = 60\frac{\pi}{180} = \frac{\pi}{3}$, to the equations gives us $v = 4\,988\,929$ and $v_{visible} \approx 831\,488$. If the highest resolution has a vertex spacing, $\Delta w = 0.5\text{m}$, this would allow us to draw the following distance in each direction:

$$d_{far} \approx 2^{l-1}\frac{(w-1)\Delta w}{2} = 2^{l-2}(w-1)\Delta w = 2^{5-2}\cdot(1025-1)\cdot 0.5\text{m} = 4096\text{m} \quad (3.7)$$

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.

3.3.2 Portability and modifiability

Since game developers are interested in reaching as many consumers as possible, they also have an incentive to support as many platforms as possible. To be usable by game developers, our generation framework needs to be able to run on many different hardware and software configurations. To target desktop and laptop computers, the implementation must at least run on Windows, Linux and OS X, and must support Intel, AMD, and NVIDIA GPUs with a wide range of driver configurations.

As stated in RQ1, the terrain representation should be portable and platform independent. For this reason, we add the additional requirement that it should also be possible to extend the framework to evaluate terrain data on other platforms as well. Such platforms include: iOS, Android, and video game consoles. Furthermore, it should also be possible to replace the framework in its entirety by creating a third-party parser and evaluator for the terrain representation format.

In Section 2.2.6, it was suggested that heightmap terrains is the most widely used terrain format. To support as many game engines as possible, the framework should at least be able to generate heightmap terrains. In Section 2.2, it was explained how heightmap terrains can be defined as a subset of other types of terrain. In fact, many of the algorithms for those types of terrain are derived directly from their heightmap counterparts. To take advantage of this, and to answer RQ3, it should be possible to extend the framework to support additional types of terrain such as voxel and vector displacement terrains.

3.3.3 List of framework requirements

Below are enumerated lists of functional and non-functional requirements for the framework. The requirements are generated from the research questions, from discussions in the previous subsections, and the desire to create a useful tool for game development.

Functional requirements

F1 A terrain preview must be updated while the terrain is edited.

F2 It must be possible to save and open terrains.

F3 It must be possible to open and generate terrains through a library.

F4 A graph interface must be provided for editing generation functions

F5 The framework must support heightmap terrains.

Non-functional requirements

Perf1 Generation must be fast enough to update a high-quality terrain preview in real-time.

Port1 Generation and editing must be possible on Linux, OS X and Windows. Generating terrains on these platforms must not rely on any third-party software being installed by the end user.

M1 It should be possible to extend the framework with evaluation on additional platforms.

M2 It must be possible to extend the editor to show previews for other types of terrains as well, such as voxel terrains and vector displacement terrains.

U1 The editor should be easy to use for a non-programmer.

3.4 Architecture

In this section, the overall architecture of the framework will be described. This will only be a high-level discussion of modules[1] and their dependencies. A discussion of the implementation of individual components will be deferred until Chapter 4.

[1]Software modules, not noise modules.

3.4.1 Framework overview

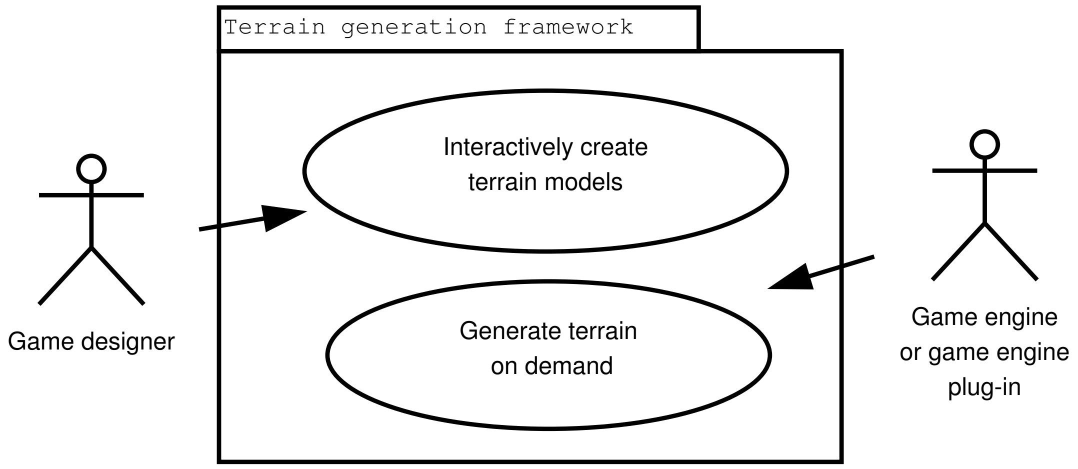

Terrain generation framework
Interactively create terrain models
Generate terrain on demand
Game designer
Game engine or game engine plug-in

Figure 3.2: Use-case diagram for the framework.

Fig. 3.2 shows a simplification of the most high-level tasks of the framework. We see the two most important stakeholders of the framework, the game designer and the game engine. The non-functional requirements of these two stakeholders are quite different. On one hand, the game designer favors usability (U1) and would like a GUI that is easy to use and runs on her developer machine. On the other hand, it is desirable for the game engine developer to keep the size of her game executable small, and limit its dependencies to support as many platforms as possible without asking the end-users to install additional software (Port1). Creating a GUI will almost certainly introduce new dependencies, which are unwanted by the game engine developer.

In order to accommodate these seemingly contradicting requirements, the framework has been divided into three parts, as shown in Fig. 3.3. The functionality needed by both the editor GUI and game engines has been factored into a library. The responsibilities of this library are discussed in Section 3.4.2.

nmlang Serialization format for function graphs (see Section 4.1).

nmlib Library for importing and exporting nmlang, manipulating and evaluating function graphs (see Section 3.4.2).

nmgui User interface for creating and editing terrain function graphs using nmlib.

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

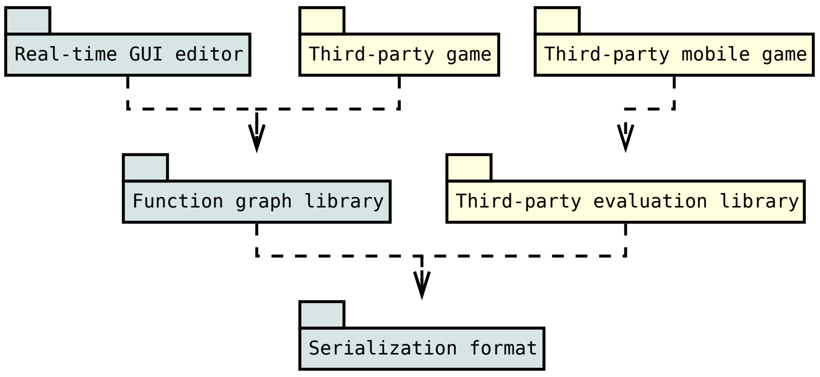


Figure 3.3: Architecture overview. The arrows show dependencies. Blue boxes are part of the framework, while yellow boxes show potential third-party software.

other heavy libraries in the GUI application. A more detailed overview of the architecture and all dependencies can be seen in Fig. 3.5.

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.

3.4.2 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. In Figure 3.4, we see the high-level use-cases of the stakeholders.

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 (Port1). 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 (Perf1). In order to achieve this kind of performance, it is crucial that the algorithm is implemented on the GPU. This is explained in Section 3.5.

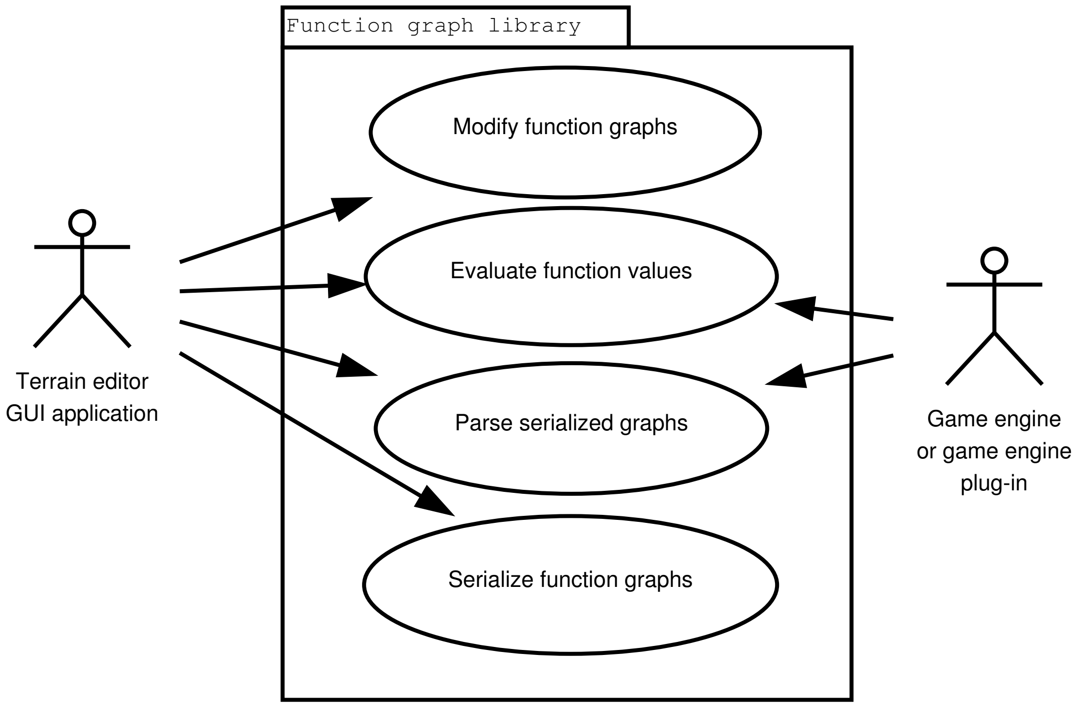


Figure 3.4: Use-case diagram for function graph library

Because the library was consumed by the GUI editor, it was desirable to be able to implement the model view controller pattern [51]. This was facilitated by implementing the observer pattern (a part of the model-view-controller pattern) on the graph model of the library. Consumers of the library can subscribe for callbacks through the header-only library "boost::signals2". This greatly simplifies integration with a great number of frameworks, including Qt (which is used for the GUI).

To further facilitate integration with other frameworks, all types in the model intended to be used with reference semantics have an additional field, a void pointer labeled "user data". This pointer can be used to store identifiers or pointers to objects wrapping the library. Such a wrapper can be seen in Fig. 3.5; the "QObject proxy" is a wrapper to make the library easier to work with in QML and JavaScript.

The library has been carefully implemented without dependencies on the GUI

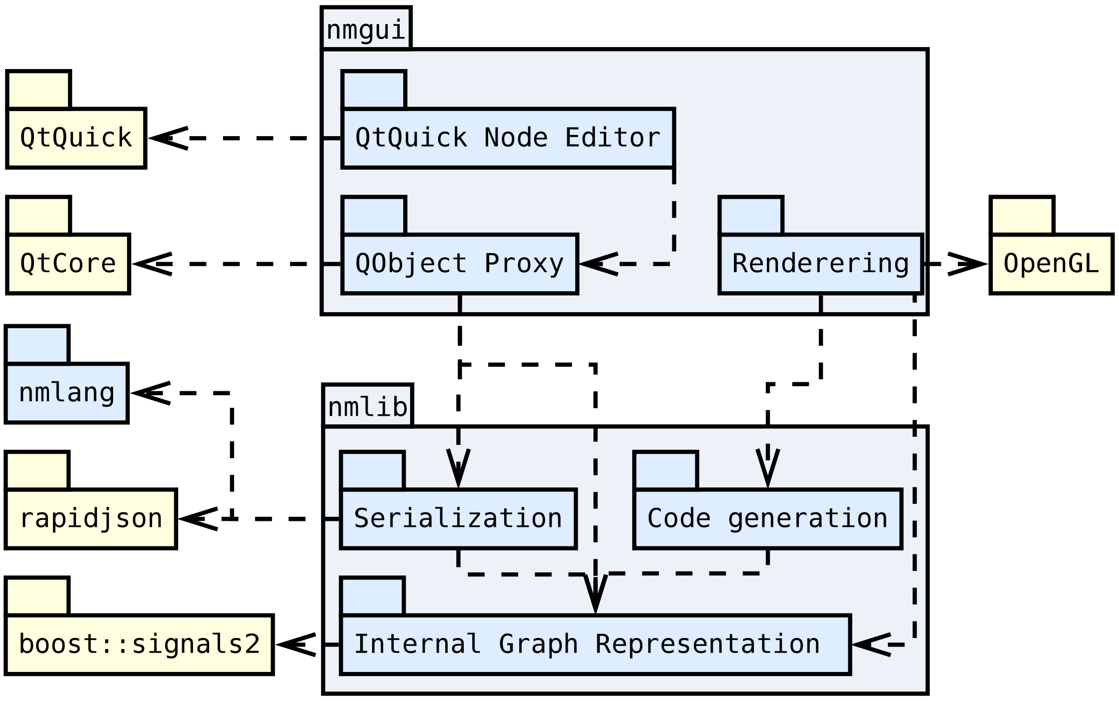


Figure 3.5: Detailed framework architecture including dependencies and package modules. The arrows indicate dependencies.

application. This means that the library can be used by game engines to parse and evaluate functions described by nmlang 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 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.

3.4.3 The Noise Modeler application

The GUI application will also be referred to as the "Noise Modeler application", or simply "Noise Modeler". There is only one stakeholder for this part of the software, and that is the game designer, the person using the application to design terrains for her games.

In line with the requirements for our framework, we want the editor to run on numerous hardware and software configurations (Port1). 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 Section 4.3 and in the user's guide (Appendix A).

3.5 Parallel computation of implicit terrains

All the CPU-based implementations of implicit stochastic terrain generation presented in Chapter 2 are too inefficient for our requirements. The benchmarks of GeoGen performed by Zábský [38] show that even with resolutions as low as 512×512 the implementation would struggle to achieve interactive performance. The popular CPU-based noise libraries, libnoise and ANL, have also been benchmarked in Section 5.1, and their performances were also insufficient.

Luckily, the algorithms for stochastic implicit terrains are clearly designed with parallelism in mind, and they are perfect candidates for an implementation on the GPU. The benchmarks performed in Section 5.1, as well as the ones performed by Zábský [38] and Olsen [30], seem to support this conclusion. The program Lithosphere is also able to achieve interactive rates by doing computation on the GPU.

In all the algorithms used in our framework, the computation at one data point is independent of the computation of neighboring points. This means that each data point can be computed in a separate thread. As long as there are more threads than points, near linear speedup can be achieved. If there are more threads than data points, however, the execution time will be bounded by the time to execute one data point, since the calculation of each data point is strictly serial.

The speedup, $S(n)$, of computing N points in parallel can thus be approximated as

$$S(n) = \frac{T(1)}{T(n)} = \left\{ \begin{array}{ll} \frac{T(1)}{\frac{T(1)}{N}} & \text{if } n > N \\ \\ \frac{T(1)}{\frac{T(1)}{n}} & \text{if } n \leq N \end{array} \right\} = \left\{ \begin{array}{ll} N & \text{if } n > N \\ n & \text{if } n \leq N \end{array} \right\} = \min(N, n) \qquad (3.8)$$

where $T(n)$ is the time required to execute with n processors. In most use cases for the library, a rather large number of data points are requested at once. In the case of providing a high-quality preview of a heightmap terrain for instance, $500\,000 < N < 1\,000\,000$ are needed. On current CPUs and GPUs, this means that $n \ll N$, resulting in a linear speedup.

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.

This leaves only OpenGL and OpenCL as viable platforms. While OpenCL might have been a good choice of platform, we took advantage of the fact that many games already make use of OpenGL for graphics. This has two advantages: Firstly, it means our framework will cause no extra dependencies or requirements for the end users (the players of the video game). Secondly, it becomes easier to directly combine computation of elevation data with rendering operations. I.e. computation of elevation data can happen directly in a vertex or tessellation shader, avoiding the need to store terrain data in memory. This feature was very useful when developing the terrain preview.

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 specifics of this process are described in Section 4.4.

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 of RQ1 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 Section 5.1 for several OpenGL implementations.

3.6 Development process

Because there was only one developer working on the project, a strict development model, such as Scrum or Kanban, was deemed inappropriate as it would impose too much overhead to be useful. Instead, an informal agile approach was taken, where focus was placed on implementing a minimal prototype as early as possible. This prototype was iteratively refined into the final product. Sub-tasks and planning were managed by a simple project backlog on Trello.com.

The library was developed using test-driven development (TDD). 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. This way it was possible to safely refactor large parts of the framework without fearing that unintended side effects might go by unnoticed. Breaking changes were usually discovered easily by failing unit tests.

Some unit tests were also written for the rendering parts of the GUI application.

One requirement of the framework is to provide an intuitive user interface, this implies that user testing is necessarily an important part of the development process. The limited time frame of the project, however, meant that there was not enough time to perform the extensive user testing that is appropriate for an end-user application. While formal quantitative user testing was sacrificed in order to speed up development time, informal and qualitative pilot testing was still carried out throughout the development process. 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.

3.7 Verification

In this chapter, solutions to several research questions have been proposed. The first point that needed to be verified, was that our representation for stochastic implicit terrain surfaces is sufficient to describe procedural terrains. As explained earlier, the implementation of our library demonstrates that the representation can be used to describe procedural terrains.

The second matter that needed to be verified, was that 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 has been performed and will be explained in Section 5.1.

A quantitative user test with a questionnaire, such as the SUS usability scale [59], would have been appropriate to judge the usability of the application. Sadly, our time-frame was limited, hence there was not enough time to arrange such a test. Although this makes it harder to discuss the usability of the system, the pilot testing provided some useful feedback. It is also an advantage that the interface is similar to offline terrain generation systems.

Initially, the plan was to develop a plug-in to a game engine in order to demonstrate how the framework might be used in game development. This was not performed, due to lack of time. However, the benchmark program does serve as a minimal tool showing that terrain generation can run without the GUI. Calculation is run for a requested patch of terrain, and the results are brought back to CPU memory, just like a plug-in would most likely do.

Chapter 4

Implementation

In this chapter, implementation details of the developed software is discussed. Section 4.1 contains a description of the serialization format designed for the terrain genotypes. An overview of the class design of our library is given in Section 4.2. In Section 4.3, the user interface is described. The details of how the terrain data is computed on the GPU is covered by Section 4.4. Finally, our terrain rendering implementation is explained in Section 4.5.

4.1 Serialization

This section describes how function graphs are serialized so that they can be stored by the editor and loaded by the game engine using our library.

4.1.1 The JSON format

Function graphs are serialized using JSON[1]. JSON is a lightweight data-interchange format that is easy for humans to read and write. It features a simple representation of name-value pairs of arrays, numbers, booleans, and strings. For a description of the JSON syntax, see Crockford [60].

There are several advantages of using JSON compared to other formats. Firstly, it is a well-supported format, with serialization and parsing libraries available for over 60 programming languages. This eases development of any potential third-party tools or evaluators for the format, making it as portable as possible. Secondly, it is a human-readable language, making it easy to understand, and even edit, by opening it in a text editor.

[1]JavaScript Object Notation

4.1.2 Noise Modeler documents

Noise Modeler documents are JSON-encoded files describing one or more user-defined functions (also called module types). When saved as files, the extension ".nm.json", is used. The first part of the file extension, ".nm", is short for noise model. The second part of the extension, ".json", is appended because this makes existing programs correctly classify the file as a JSON-document. This means that the file is editable by specialized JSON editors, and that text editors will use the correct syntax highlighting.

The structure of an nm.json-document will be described in detail in Sections 4.1.3 to 4.1.4

4.1.3 Module types

The root object of an nm.json-document has only one member, `moduleTypes`. This is an array of different types of nodes available. Each entry in `moduleTypes` describes a custom function. One entry in the array may depend on another; the entries therefore have to be topologically sorted according to their dependencies.

The description of each module type has five attributes:

name (string) a unique identifier for the type.

description (string) an informal description of what the type does.

inputs an array of inputs that the module accepts. Each entry has up to three key-value pairs.

> **name** (string) and identifier for the input.
>
> **type** (string) the type of this input. It takes the form:
>
> $$\langle dimensionality \rangle f$$
>
> The dimensionality is a number that describes the dimensionality of the input (vector) the "f" indicates, that this is a floating point vector (the only supported type so far). For example, a three-dimensional vector of floating point values would be denoted as `3f`.
>
> **value** an optional default value for the input. If not specified, the value will be set to zero.

modules (array) a graph of modules, and how their inputs and outputs are connected.

outputs an array of outputs (their external name, and their corresponding name in the internal `modules` graph).

4.1.4 Modules

The `modules` member of a module type is an array of instantiated module types. Each entry describes the name of the module, an optional description, which module type this is an instance of, and which outputs (if any) the inputs are connected to, or whether the input is bound to a constant.

name (string) a unique identifier for the module.

type (string) an identifier for the module type of this module.

description (string) an optional informal description of what the module does.

inputs an array of string-string pairs; the keys refer to inputs defined in this module's module type, the values refer to the outputs of other modules. The values take the form

$$\langle moduleName \rangle . \langle outputName \rangle$$

The values in the `inputs` list may optionally be specified as numbers or arrays of numbers to bind the input to a constant rather than the output of another module.

If an input is not present in the array, its default value from the module type definition will be used.

4.1.5 Serialization example

Here is an example of how a terrain function can be serialized:

Listing 4.1: Example terrain function

```
{
    "moduleTypes": [
        {
            "name": "terrainHeight",
            "description":"determines elevation based on position",
            "inputs": [
                {
                    "name": "pos",
                    "type": "2f"
                }
            ],
            "outputs": [
                {
                    "name": "height",
                    "source": "add1.out"
                }
            ],
            "modules": [
                {
                    "name": "fbm1",
                    "type": "fbm",
                    "inputs": {
                        "pos": "inputs.pos"
                    }
                },
                {
                    "name": "add1",
                    "type": "add",
                    "inputs": {
                        "lhs": "fbm1.out",
                        "rhs": "-3"
                    }
                }
            ]
        },
        {
            "name": "groundColor",
            "description":"determines ground color based on elevation",
            "inputs": [
                {
                    "name": "height",
                    "type": "1f"
                }
            ],
            "outputs": [
                {
                    "name": "color",
                    "source": "mux31.out"
                }
            ],
            "modules": [
                {
                    "name": "mul1",
                    "type": "mul",
                    "inputs": {
                        "lhs": "inputs.height",
                        "rhs": "2"
                    }
                },
                {
                    "name": "mux31",
                    "type": "mux3",
                    "inputs": {
                        "x": "0.5",
                        "y": "mul1.out",
                        "z": "mul1.out",
                    }
                }
            ]
        },
        {
            "name": "terrain",
            "description":"determines terrain color and elevation based on ↩
                position",
            "inputs": [
                {
                    "name": "pos",
                    "type": "2f"
                }
            ],
            "outputs": [
                {
                    "name": "color",
                    "source": "terrainColor1.color"
                },
                {
                    "name": "height",
                    "source": "terrainHeight1.height"
                }

            ],
            "modules": [
                {
                    "name": "terrainHeight1",
                    "type": "terrainHeight",
                    "inputs": {
                        "pos": "inputs.pos"
                    }
                },
                {
                    "name": "terrainColor1",
                    "type": "terrainColor",
                    "inputs": {
                        "height": "terrainHeight1.height",
                    }
                }
            ]
        }
    ]
}
```

The rationale behind having multiple user-created module types is two-fold: Firstly, it makes it possible to create abstractions for particularly complicated parts of a graph, by labeling the inputs and encapsulating the sub-graph. Secondly, it makes it easy to parametrise and reuse commonly occurring sub-graphs.

For example, working with a “terrainHeight” module with a “roughness” input may be more intuitive than working with an “fBm” module with “octaves”, “lacunarity”, and “gain”.

In Listing 4.1 there are three module types: `terrainHeight`, `groundColor`, and `terrain`. The `terrain` type depends on `terrainHeight` and `groundColor`, while `terrainHeight` and `groundColor`, on the other hand, are independent.

`terrainHeight` is a user type that accepts a single input, `pos`; a two-dimensional vector representing the width and length position. A single output, `height`, gives the terrain height at that position.

This user type is instantiated inside the `terrain` user type, which accepts a position as input and gives a height and a color as output. Inside `terrain` the output of a `terrainHeight` module is used as the input for a `groundColor` module, which calculates a color based on a height.

4.2 Library design

In this section, we will discuss the class design of "nmlib", the library part of the framework. We will also give an overview of the core classes of the library and the most important design patterns used.

A class diagram of the library can bee seen in Fig. 4.1. In order to save space, the class diagram has been simplified in many ways. First, the library is const-correct[2], and consequently many methods have both a const and a non-const version; these are shown as one in the class diagram. Furthermore, most data members are omitted from the diagram, unless they help portray the purpose of a class. Many classes from the code generation module are also omitted, as this module will be described more thoroughly in Section 4.4.1.

Although this is not shown in the diagram, all classes are members of the `nm` namespace. This is done to prevent naming conflicts with other libraries and game engine code.

A completely accurate and detailed description aimed at the users of the library has been written and generated using Doxygen. Doxygen is a tool that parses C++ header files and special comment markup in order to create a documentation web page. Doxygen can also be used to create a reference manual for printing. Online documentation can be found at `http://docs.noisemodeler.org`, while an excerpt from the reference manual is included in Appendix D.

The class design of the library is closely related to the domain model presented in Section 4.1 and Section 3.2. For instance, there are separate classes corresponding to modules, module types, signals, signal types, and graphs. Inputs and outputs, however, are a special case. There are two sets of classes describing inputs and outputs. The `ModuleInput` and `ModuleOutput` classes describe inputs and outputs of a module type, i.e. signal type information, and a default value for the input. The `InputLink` and `OutputLink` classes, on the other hand, describes information related to a module (an instantiated module type). They contain a reference to the `ModuleInput` or `ModuleOutput` they are representing, along with additional information specific to the module they are attached to. Such information include whether the input is connected to an output, or if it should be assigned a specific constant value.

The `TypeManager` class is a class managing all available module types. It may be populated with built-in types, using the `initBuiltinTypes` method. Additional module types may also be added to the type manager. The type manager is the top-most level of the model hierarchy. When documents are parsed, a type manager is returned, containing all the parsed module types.

[2]Const-correctness is a C++ paradigm where objects of a class may be declared immutable (const). It will then be a syntax error to call methods that are not declared as "const", meaning they do not change the object.

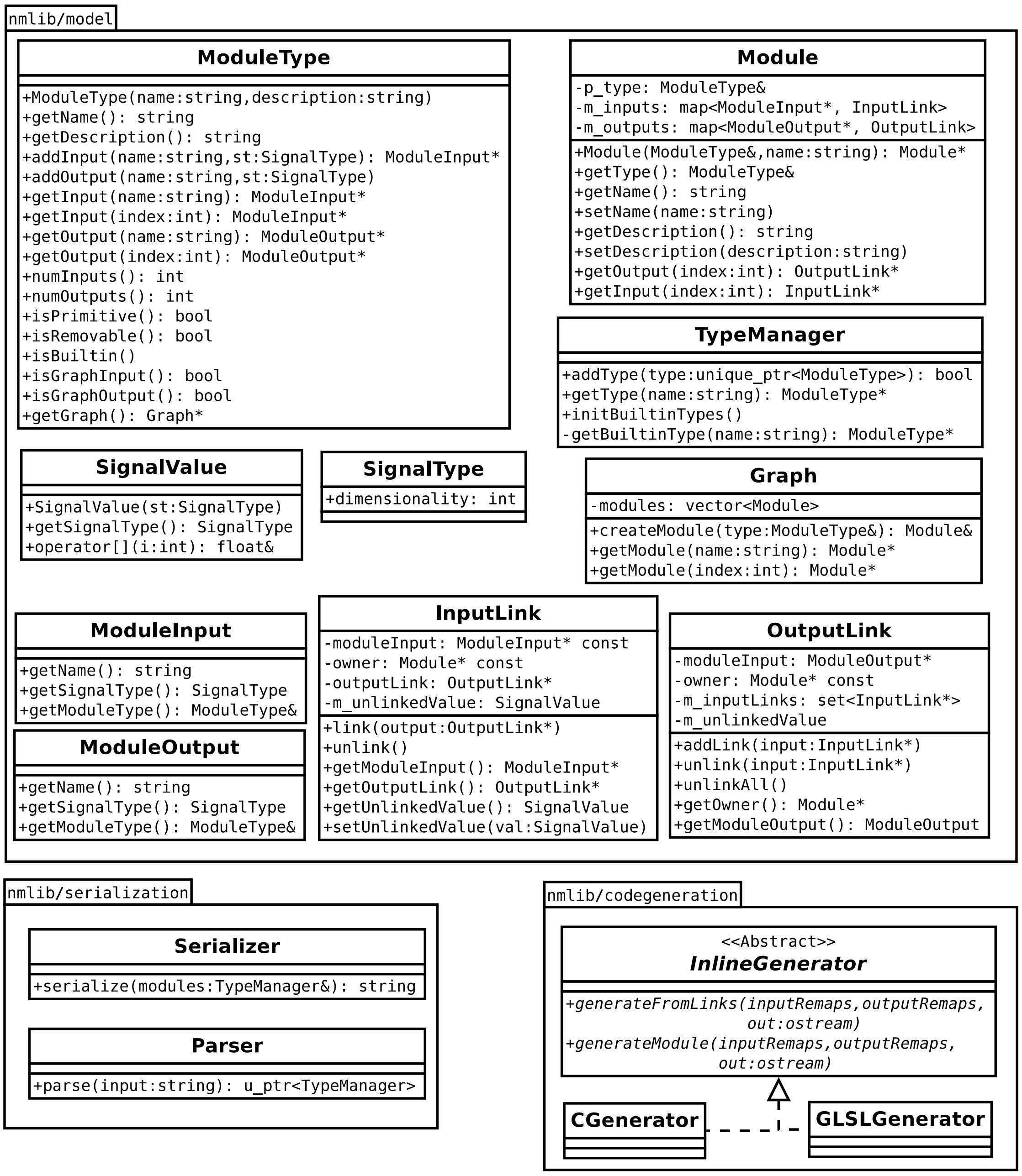

nmlib/model
ModuleType
+ModuleType(name:string,description:string)
+getName(): string
+getDescription(): string
+addInput(name:string,st:SignalType): ModuleInput*
+addOutput(name:string,st:SignalType)
+getInput(name:string): ModuleInput*
+getInput(index:int): ModuleInput*
+getOutput(name:string): ModuleOutput*
+getOutput(index:int): ModuleOutput*
+numInputs(): int
+numOutputs(): int
+isPrimitive(): bool
+isRemovable(): bool
+isBuiltin()
+isGraphInput(): bool
+isGraphOutput(): bool
+getGraph(): Graph*
Module
-p_type: ModuleType&
-m_inputs: map<ModuleInput*, InputLink>
-m_outputs: map<ModuleOutput*, OutputLink>
+Module(ModuleType&,name:string): Module*
+getType(): ModuleType&
+getName(): string
+setName(name:string)
+getDescription(): string
+setDescription(description:string)
+getOutput(index:int): OutputLink*
+getInput(index:int): InputLink*
TypeManager
+addType(type:unique_ptr<ModuleType>): bool
+getType(name:string): ModuleType*
+initBuiltinTypes()
-getBuiltinType(name:string): ModuleType*
SignalValue
+SignalValue(st:SignalType)
+getSignalType(): SignalType
+operator[](i:int): float&
SignalType
+dimensionality: int
Graph
-modules: vector<Module>
+createModule(type:ModuleType&): Module&
+getModule(name:string): Module*
+getModule(index:int): Module*
ModuleInput
+getName(): string
+getSignalType(): SignalType
+getModuleType(): ModuleType&
ModuleOutput
+getName(): string
+getSignalType(): SignalType
+getModuleType(): ModuleType&
InputLink
-moduleInput: ModuleInput* const
-owner: Module* const
-outputLink: OutputLink*
-m_unlinkedValue: SignalValue
+link(output:OutputLink*)
+unlink()
+getModuleInput(): ModuleInput*
+getOutputLink(): OutputLink*
+getUnlinkedValue(): SignalValue
+setUnlinkedValue(val:SignalValue)
OutputLink
-moduleInput: ModuleOutput*
-owner: Module* const
-m_inputLinks: set<InputLink*>
-m_unlinkedValue
+addLink(input:InputLink*)
+unlink(input:InputLink*)
+unlinkAll()
+getOwner(): Module*
+getModuleOutput(): ModuleOutput
nmlib/serialization
Serializer
+serialize(modules:TypeManager&): string
Parser
+parse(input:string): u_ptr<TypeManager>
nmlib/codegeneration
<<Abstract>>
InlineGenerator
+generateFromLinks(inputRemaps,outputRemaps, out:ostream)
+generateModule(inputRemaps,outputRemaps, out:ostream)
CGenerator
GLSLGenerator

Figure 4.1: Class diagram for the library.

A graph is an optional member of a module type. This field is only used for user types, i.e. types that are described by a module graph themselves.

There is almost no inheritance in the library. This is a result of three factors. First composition has been favored over inheritance, as this often leads to cleaner code. Second, the requirements of the library favors dynamic design pattern that delay decisions until run-time. Third, it is easier to wrap the library in other languages and frameworks if it does not rely on polymorphism, as it may not be supported by the target language.

The exception to this rule is the `UserData` class (not shown in diagram). `Graph`, `InputLink`, `OutputLink`, `Module`, `ModuleType`, `ModuleInput`, `ModuleOutput` and `TypeManager` inherit from this class. This is done to limit the amount of boiler plate code in the library. This class provides a `void` pointer which can be used by calling code to attach data to an object. In the GUI application, this field is used to store references to corresponding `QObject` wrapper objects. Another class, `NonCopyable`, serves a similar purpose to disable copy and assignment constructors with a limited amount of boilerplate code.

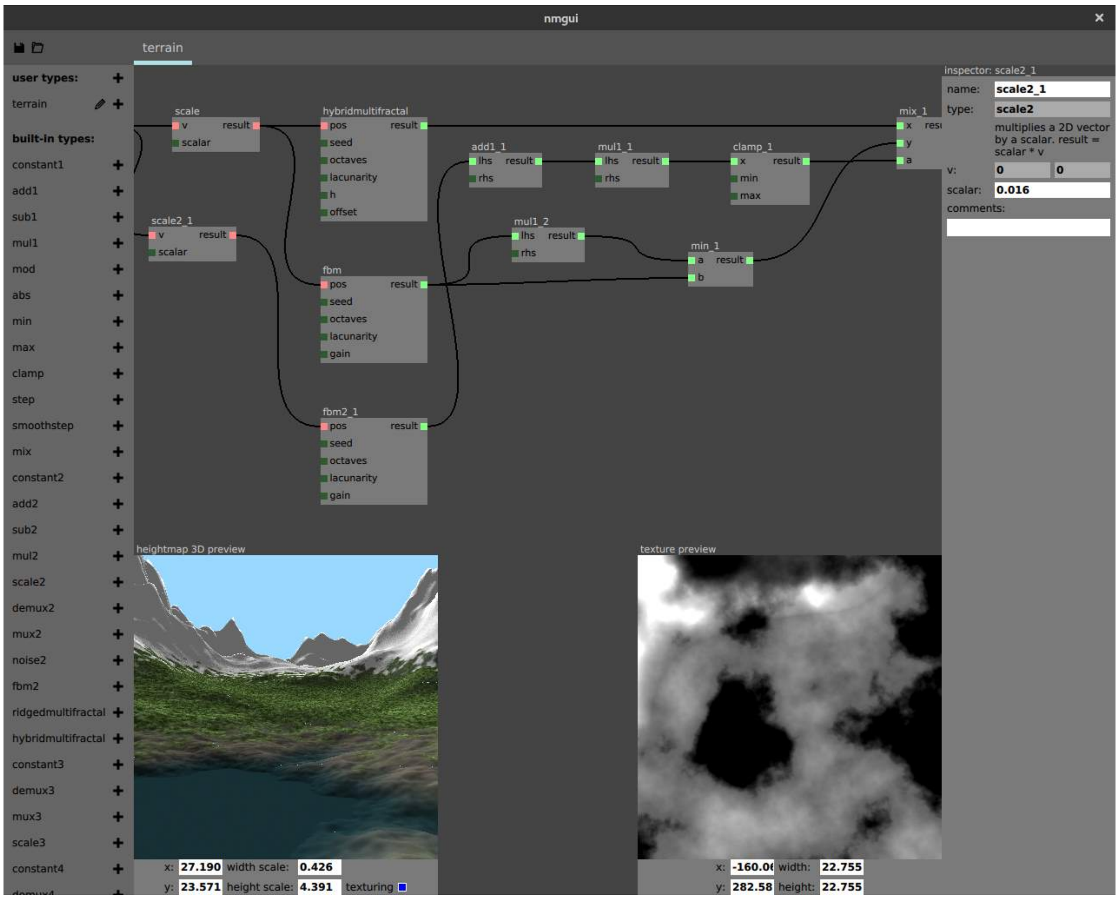
As mentioned in Section 3.4.2, callbacks are provided through boost::`signals2`. For each data member with public accessors in the model module, there is a corresponding signal. It is possible to subscribe to a callback function by calling the `connect` function of a signal with a function with a matching signature. In each callback, a reference to the owning object is included. These callbacks are mostly used by external code to react to changes in the model. In the GUI application, the views are updated appropriately. The signals are also used internally to make sure the model is always correct. For example: If a new input is added to a module type, a corresponding input is added to all modules of that module type.

In several places, the factory method pattern is used. `Graph` has a factory method for creating `Modules` that belong to the graph. `ModuleType` has factory methods for creating new inputs and outputs. `TypeManager` has factory methods for creating new `ModuleTypes`.

4.3 User interface

Below follows a discussion of the most important elements of the user interface of the Noise Modeler application. An in-depth user's guide including a user interface description can also be found in Appendix A. The class-level implementation details of the user interface will not be discussed since it would make the description too long.

A screenshot of the implemented user interface can be seen in Fig. 4.2. Additionally, a demonstration video is available on the project website:
`http://www.noisemodeler.org/demo.html`



inspector: scale2_1
name: scale2_1
type: scale2
multiplies a 2D vector by a scalar. result = scalar * v
v: 0 0
scalar: 0.016
comments:

x: 27.190 width scale: 0.426
y: 23.571 height scale: 4.391 texturing

x: -160.0(width: 22.755
y: 282.58 height: 22.755

Figure 4.2: A screenshot of the Noise Modeler application.

4.3.1 Graph editor

The graph editor is the center of attention of the application. Modules are shown as boxes labeled with a name. On the left side of each box, there is a list of inputs, as well as a square indicating the type of input (i.e. the dimensionality of the signal). There is also a similar list of outputs on the right side of the box. Each box may be dragged around the area of the graph. It is possible to connect an output to an input by dragging from the color-coded box on the right side of a module to a box of the same color on another module. After doing this, a Bézier curve will be drawn from the input module to the output module to represent the connection.

For heightmap terrain, there will be two special modules, labeled “inputs” and “outputs”. The “inputs” module has only one output, “pos”, this is the input position to the function when used to query for heights. Similarly, there is only one input in the “outputs” module, “height”, which will be assigned as the output of the height function.

This interface is very similar to the interface of World Machine, which is shown in Fig. 2.12, and to procedural shader editors. The major difference, however, is that, in this implementation, all inputs to a function are explicit and visible, while in World Machine, some outputs are implicit, such as the position input to noise generators. The reason for doing this, is flexibility. For instance, it may be useful to be able to scale or transform the position signal.

4.3.2 Inspector

The inspector is a GUI component located along the right side of the application. It is used to edit the properties of modules not shown in the graph editor. The primary function of the inspector is to assign constant values to disconnected inputs. For instance, a multiplication module may only have one connected input, while the other input may be assigned a constant through the inspector. The inspector may also be used to change names and descriptions of modules.

Instead of making a separate GUI component for module customization, the functionality could have been included within the graph editor itself. The reason for keeping it separate, is that it allows the graph editor to stay more compact, thus allowing more modules to fit on the screen at once. Keeping the functionality within the module inspector, also makes it possible to show a maximized preview in the main area of the application, while still being able to change the constant values in the inspector.

4.3.3 List of module types

To the left (see Fig. 4.2) is a list of module types that may be added to the graph. The list is divided into two categories. "User types" are module types created by the user, while "built-in types" are defined by the framework.

Clicking the plus symbol to the right of a module type will add it to the graph. There are also buttons for editing and adding new user types.

4.3.4 Real-time previews

There are two real-time previews of the heightmap terrain function: A 3D heightmap terrain preview, and a texture preview. These previews are updated instantly according to changes performed in the graph editor or inspector. The previews will be discussed in Section 4.5.

4.3.5 Intended workflow

Here is the intended workflow when designing terrain for a game.

1. Design different terrain types, or biomes, as separate user types.
2. Expose useful or intuitive parameters as inputs to the user type.
3. Create a composite terrain by mixing the user types defined earlier in a new user type, name this user-type “terrain”.
4. Save the terrain as a “nm.json” document.
5. Load the “nm.json” document using a plug-in for a game engine.

4.4 GPU evaluation

While Section 3.5 gave an overview of how terrain can be evaluated in parallel using the GPU, this section describes the specifics of how the GLSL code generation was implemented. Section 4.4.1 describes how GLSL code was generated.

Note that while we chose to develop code generation for GLSL, the framework is still designed to allow computation on other platforms. Therefore, it should be trivial to port the GLSL code generation module to platforms with a similar syntax (such as C, CUDA or OpenCL), as the code has been written with portability in mind.

Portability was also important when choosing an OpenGL version. While the latest version, OpenGL 4.4, offers many useful features, they are not available on all platforms. Specifically, the open-source Linux drivers only support OpenGL 3.3. Furthermore many graphics cards do not support newer OpenGL versions because of hardware limitations. Consequently, Noise Modeler was limited to using OpenGL 3.0 features only. Because of this, the program will run on almost all combinations of graphics cards, operating systems and driver versions.

4.4.1 Generating GLSL code

This section describes how GLSL code generation was implemented. The implemented technique for translating from a function graph to GLSL code will now be presented.

Generating code given InputLinks and OutputLinks

In order to make use of a function graph, it is necessary to have a way to compute output values given certain input values. To be flexible, we want to be able to specify which inputs we are interested in changing, and which outputs we want to use. This will allow us to leave out computation for the parts of the graph that are not required for the outputs we are interested in.

The library provides the following function:

```cpp
void InlineGenerator::generateFromLinks(const std::vector<↩
    InputRemap> &inputRemaps, const std::vector<OutputRemap> &↩
    outputRemaps, std::ostream &out);
```

This function generates inline code for parts of a function graph, by writing to the provided std::ostream& output parameter. It takes as input a collection (an std::vector) of "InputRemaps", and a collection of "OutputRemaps". "InputRemap" and "OutputRemap" are defined as follows:

```cpp
struct InputRemap {
    std::string externalName;
    InputLink* inputLink;
};
struct OutputRemap {
    std::string externalName;
    OutputLink* outputLink;
};
```

"InputRemap" and "OutputRemap" contain information about how the generated code should interface with surrounding code. The "externalName" member, is the name of the input or output in code surrounding the generated code, while the "*link" members describe the corresponding inputs and outputs of modules in our graph. This means the "externalName" in an "InputRemap" has to be declared and assigned in code preceding the generated code, while the "externalName" of an "OutputRemap" is declared and assigned in the inline code, available to be used by surrounding code afterwards.

This is a very flexible approach, since it makes it possible to prune parts of a graph that are irrelevant to our given inputs and outputs, while it may also easily be used to provide a way to generate function definitions for user defined module types including all their inputs and outputs.

The generated code has the following structure:

Listing 4.2: The structure of code generated by generateFromLinks

```
//<declare a variable name externalName for each OutputRemap>
//i.e. float height;
{
    //<internal code for computing the outputs>
    //i.e. float nm_id0 = 3+2;

    //<for each OutputRemap, assign a computed value to its ``externalName''>
    //i.e. height = nm_id0;
}
```

The generated code is wrapped inside a scope. This means that the code will not interfere with or use the surrounding code, except through the specified variables.

We will now describe how the "internal code for computing outputs" in Listing 4.2 is generated.

First, a sorted list of required modules is generated by traversing the graph topologically. The traversal starts at the outputs and follows graph edges from inputs to outputs. While this is happening, the traversal algorithm manages a list of visited modules to prevent visiting the same node twice. Whenever one of the inputs which have a corresponding "InputRemap" is reached, the traversal is blocked. The output of this traversal is a list of all visited modules, in the order they have to be computed.

This list is subsequently iterated over and code is generated for each module.

Generating code for modules

Again, the code generated for each module is wrapped within a scope, while collections of "InputRemaps" and "OutputRemaps" are maintained to direct the output of one module to the input of another.

Listing 4.3: GLSL generated for a 2D fBm module

```
//////////////////////////////////////////
//Generating code for module: "fBm" of type "fbm2"
//////////////////////////////////////////

//output declarations
float nm_id_3;
{
   //input declarations
   vec2 pos;
   float seed;
   float octaves;
```

```
    float lacunarity;
    float gain;

    //assign unlinked values for inputs
    pos = vec2(0, 0);
    seed = 0;
    octaves = 10;
    lacunarity = 2;
    gain = 0.5;

    //reassign any connected inputs
    pos = nm_id_2;

    //funtion body for module "fBm" of type "fbm2"
    float result = fbm2d(pos, octaves, lacunarity, gain, seed);
    //end function body

    //output assignments
    nm_id_3 = result;
}
//////////////////////////////////////////
//end of module: "fBm" of type "fbm2"
//////////////////////////////////////////
```

Listing 4.3 gives an example of code generated for a two-dimensional fBm module. First is a comment that shows what module we are generating code for.

After that, the remapped outputs are declared. We can see that it is declared as “nm_id_3”; this identifier has been generated by a call to “InlineGenerator::getUniqueId()” which generates a unique id with the prefix “nm_id_”. This id will be used later by surrounding code, perhaps as input to another module.

After that, the rest of the code is wrapped in a scope.

A list of variable declarations have been generated from the list of “ModuleInputs” located in the type of the module.

Assignments to these declared variables are later generated by iterating through the “InputLinks” of the module, which contains an “unlinked value”, i.e. the value an input will have unless it is connected to the output of another module. If an input is connected, it will be reassigned according to the lists of remaps. In this example, we can see that the variable “nm_id_2” has been assigned to the input “pos”.

The function body is specific to each module type. In this case, it is simply a call to a previously declared “fbm2d”-function. Had this been an “add” module, the body might simply have been “float result = lhs + rhs;”. Note that this is the only part that is generated differently for each module type. All other code is generated from information available in the model.

At last, the output is assigned to the external output.

4.4.2 GLSL Noise implementation

Since it is desirable to create standalone GLSL code that does not rely on texture lookups, the approach in Green [44] can not be used. Instead, our implementation is based on the source code of McEwan, Sheets, Richardson, *et al.* [45] (see Section 2.9.8), which is released under the MIT license.

A shortcoming of their implementation is that it is not possible to supply a seed to the algorithm in order to get a different pseudo-random function. This is unacceptable for our requirements, since generating a new terrain for each session is one of the main benefits of using procedural generation in games.

This problem is present because the implementation uses permutation polynomials rather than permutation arrays (which can easily be seeded). A different permutation polynomial could be selected, but only a limited number of polynomials exist for the desired range.

The issue has been pointed out in the issue tracker of the project[3]. One of the authors, Stefan Gustafson, responded by proposing two different solutions to the problem:

- Simply add an offset to the input of the function. The domain of the domain of the function is extremely large and adding a large offset could do what we want. The problem, however, is that if the offsets are not large enough, we might risk that players of a game can randomly encounter terrain they recognize from a previous session. This is also undesirable, since floating point errors start occurring when the offset gets too large.

- Run an additional permutation on an N-dimensional seed vector before looking up the lattice gradient, where N is the dimensionality of the noise function. This means there will be 282^N different seeds possible (the range of the permutation polynomial being 289). This solution has a bigger performance impact than simply adding an offset.

In our implementation, the second approach was used. For heightmap terrains, which use 2D-noise, this means that the number of possible seeds is limited to $289^2 = 83\,521$. To further increase the number of seeds available, it would have been possible to use a different permutation polynomial with a larger range, but this was not done, as 83 521 seeds are more than enough for our purposes.

4.4.3 Implementing fBm and multi-fractal algorithms

Once a good noise function was implemented, it was easy to port the fractal-based heightmap generating functions: fBm, hybrid multifractal and ridged multifractal.

[3] `https://github.com/ashima/webgl-noise/issues/9`

Below is the implementation of the fBm:

Listing 4.4: GLSL implementation of fBm

```glsl
float fbm2d(vec2 pos, float octaves, float lacunarity, float gain, float seed) {
    float sum = 0;
    float amplitude = 1;
    float frequency = 1;
    for(int i=0; i<octaves; ++i){
        sum += snoise(pos*frequency, seed)*amplitude;
        amplitude *= gain;
        frequency *= lacunarity;
    }
    return sum;
}
```

The code of Musgrave [3] for hybrid multifractals and ridged multifractals was a bit more complicated to port to GLSL, as both algorithms have a strictly serial computation step for pre-computing the spectral weights that are executed only the first time the algorithm is run.

The spectral weight, w_s, for each individual octave, i, is computed using the equation

$$w_s = (l^i)^{-H} \tag{4.1}$$

where H is the highest fractal dimension, and l is the lacunarity. The spectral weights could have been computed as a step of the generation, thus achieving the same optimization as Musgrave [3]. Instead, we chose to repeat the computation of the spectral weights for each point. This has a negative impact on performance, but means that the fractal dimension, H, does not need to be constant. It also simplifies the code significantly.

Below is a listing of the code for our GLSL port of the ridged multifractal terrain algorithm. In line 23, the spectral weight from Eq. (4.1) is computed directly for each point.

Listing 4.5: GLSL implementation of ridged multifractal terrain

```glsl
1  float ridgedmultifractal(vec2 pos, float octaves, float lacunarity, float h, float offset, float gain, float seed){
2
3      //compute first octave
4      float signal = snoise(pos, seed);
5      signal = abs(signal);
6      signal = offset - signal;
7      signal *= signal;
```

```
    float result = signal;

    //compute remaining octaves
    float frequency = lacunarity;
    float weight = 1;
    for(int i=1; i<octaves; ++i){
        weight = signal*gain;
        weight = clamp(weight, 0, 1);
        signal = snoise(pos*frequency, seed);
        signal = abs(signal);
        signal = offset - signal;
        signal *= signal;
        signal *= weight;

        //compute spectral weight
        float exponent = pow(pow(lacunarity, i), -h);

        //add the contribution from this octave to the result.
        result += signal * exponent;

        weight *=  signal; // update the monotonically ↩
           decreasing weight
        frequency *= lacunarity;
    }
    return result;
}
```

As the implementation for hybrid multifractal terrains is very similar to that of ridged multifractal terrains, its GLSL code will not be listed here.

4.5 Rendering terrain previews

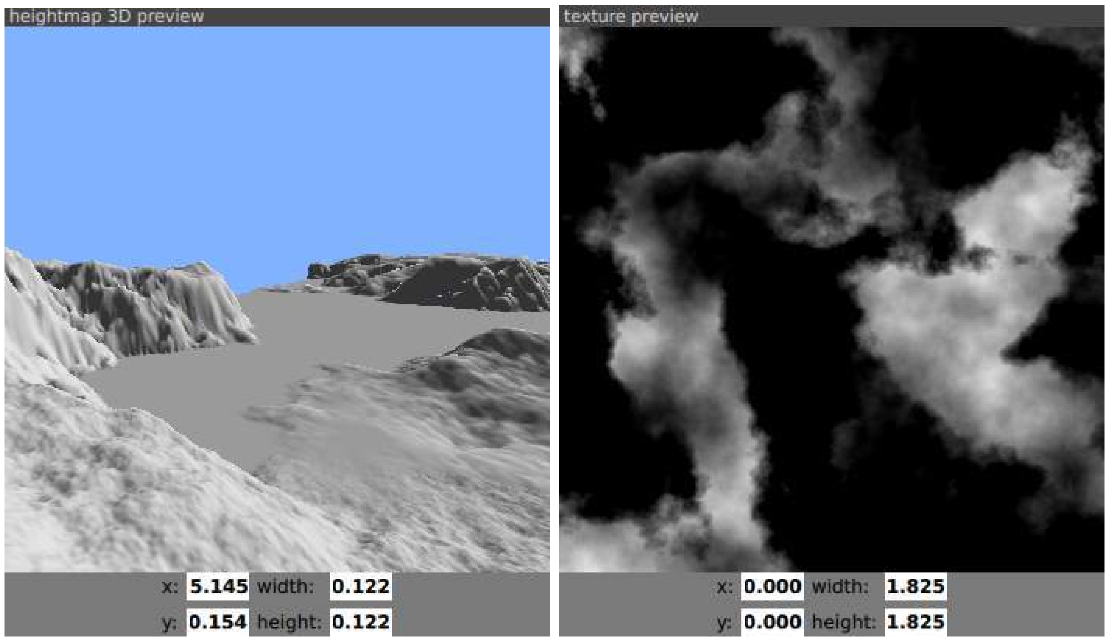
Two different types of real-time previews have been implemented. They will be described in the following subsections.

4.5.1 Texture preview

The texture preview is used to show the output of a two-dimensional function. Values between 0 and 1 are mapped to colors between black and white. When modeling terrain, this acts as a map where the tallest areas are shown as bright white, while the lowest are shown as completely black. This resembles the way heightmaps usually look when opened by an image manipulation program.

The domain of the patch sampled can be manipulated similarly to the way web applications usually allow their users to browse maps:

- Mouse-wheel scrolling scales the domain
- Click-and-dragging translates the domain



(a) 3D preview (b) Texture preview

Figure 4.3: Screenshots of heightmap previews

The texture preview is implemented by drawing a triangle strip with two triangles forming a rectangle. The function to be previewed is exported as a GLSL-function (described in Section 4.4.1) which is then sampled in the fragment shader used to draw the triangle strip.

The vertex data, which is constant, is used to help the vertex shader differentiate between the different corners of the rectangle. I.e. (1, 1) is used to represent the top-right corner. This value is passed through and used as the position within the framebuffer. (1,1) is also the top right of the framebuffer.

The current domain of the function to be rendered is transferred to the shader program through a single uniform value, “domain”, of type “vec4”. This uniform value is used to scale and offset the vertex coordinates to get the function coordinates to sample in the current corner. This coordinate is then passed on as the output of the vertex shader. The coordinates are then interpolated by the OpenGL pipeline before arriving in the fragment shader.

Listing 4.6: Vertex shader for texture preview

```
#version 130
uniform vec4 domain; //{x, y, width/2, height/2}
in vec2 vertices;
out vec2 coords;
void main() {
    gl_Position = vec4(vertices.x,vertices.y,0,1);
    coords = vertices.xy*domain.zw + domain.xy;
}
```

Listing 4.7: Generation of fragment shader for texture preview

```
std::stringstream fs;
fs << "#version 130\n";
//dynamically generate ``elevation'' function
fs << getHeightFunctionSource();
fs << ""
      "in vec2 coords;\n"
      "void main() {\n"
      "    float height;\n"
      "    elevation(coords, height);\n"
      "    gl_FragColor = vec4(height, height, height, 1);\n"
      "}\n";
```

In the fragment shader, the terrain function is called using the linearly interpolated coordinates, which the vertex shader computed in each corner. The result of the terrain function is then multiplied with pure white before being assigned as the fragment color.

The result is that heights below zero appear as black, while heights above 1 appear as pure white. The values in between are represented by various shades of gray.

4.5.2 Heightmap 3D preview

Another type of preview is the heightmap 3D preview. In this preview, a two-dimensional height function can be viewed as a 3D-terrain from a perspective camera, which can be controlled by a keyboard and a mouse (controls and user interface components are described in Appendix A).

The terrain preview is rendered to a framebuffer object using C++ and OpenGL. The framebuffer object is then handed over to Qt Quick and drawn into the GUI inside the “heightmap 3D preview” window (see Section 4.3).

The following subsections will describe the details of how C++ and OpenGL are used to render into the framebuffer object.

LOD algorithm

Terrain is rendered using a level-of-detail (LOD) algorithm roughly based on the algorithm of Strugar [49], described in Section 2.10.3. The algorithm was selected because of its ability to handle large changes in altitude. This was a natural choice, since one of the features of the terrain preview is the ability to fly around freely. Designers might want to look at the terrain, both from ground level and from a top-down perspective. The algorithm is also fairly flexible, and the most intensive calculations can run in hardware even on old versions of OpenGL.

The full algorithm was not implemented because other features had to be prioritized. Consequently, our implementation is simplified to not interpolate between different LOD-levels, for this reason, the preview has visible seams and popping effects when LOD-levels change. These seams can be seen in Fig. 5.1, where a bit of the light-blue sky color is visible where the seams do not match.

Using CDLOD made it possible to render a relatively large patch of terrain.

Rendering patches

After the LOD-algorithm has selected which terrain patches should be drawn at what LOD-levels, the right calls to OpenGL must be issued to draw each patch.

During the initialization step of the application, a single vertex buffer is created. This vertex buffer consists of the 2D positions needed to draw a tessellated plane using triangle strips. Coordinates range from 0 to 1 in both dimensions. The vertex buffer has no height information, since the height is computed by the vertex shader.

Most of the remaining rendering is done through OpenGL vertex and fragment shaders. Each terrain patch is drawn with a single call to “glDrawArrays”, which draws each patch as a single triangle strip.

In the vertex shader, the coordinate system is transformed so that the triangle strip covers the area corresponding to the LOD-level and world position of the current terrain patch. The final height coordinate of each vertex is then computed by sampling the terrain height function.

Normals are also computed by the vertex shader by sampling the additional heights of two neighboring vertices and taking a simple cross-product. This is a very rough estimate, and better-looking results could have been achieved by taking additional samples, computing a more accurate normal. This would have a negative performance impact, however, but would be less noticeable if the values were cached.

Listing 4.8: Vertex shader for 3D preview

```
#version 130

```

```
// <generated elevation function here>

in vec2 vertex;

out highp vec2 coords;
out vec3 normal;

uniform mat4 modelViewMatrix;
uniform mat3 normalMatrix;
uniform mat4 projectionMatrix;
uniform mat4 mvp;
uniform vec3 scaling;
uniform vec2 patchOffset;
uniform vec2 sampleOffset;
uniform float patchSize;

//simplifies getting a scaled height from the generated ↩
    elevation function
float sampleHeight(vec2 pos){
    float height;
    elevation(sampleOffset+pos*scaling.xy,height);
    return height*scaling.z;
}

void main() {
    vec2 vertexCoords = patchOffset + vertex*patchSize;
    float height = sampleHeight(vertexCoords);
    float resolution = 64;

    //compute the normals here
    float delta = patchSize/resolution;
    float rightHeight = sampleHeight(vertexCoords + vec2(delta↩
        ,0));
    float upHeight = sampleHeight(vertexCoords + vec2(0,delta));
    vec3 rightVector = normalize(vec3(delta, 0, rightHeight-↩
        height));
    vec3 upVector = normalize(vec3(0, delta, upHeight-height));

    //compute normal purely based on these two points
    vertexNormal = normalize(cross(rightVector, upVector));
    normal = normalize(normalMatrix * vertexNormal);

    vec3 vertexPosition = vec3(vertexCoords,height); //world ↩
        position
    gl_Position = mvp * vec4(vertexPosition,1); //final ↩
        projected position
}
```

The rest of the rendering is fairly simple. A Phong shading model is used.

Positions and normals computed by the vertex shader are interpolated before being picked up by the fragment shader. In the fragment shader the final color is computed by using a single hard-coded directional light, and hard-coded diffuse and ambient material colors.

Listing 4.9: Simple fragment shader for 3D preview

```glsl
#version 130

in highp vec2 coords;
in vec3 normal;

uniform mat4 modelViewMatrix;
uniform mat3 normalMatrix;

void main() {
    vec3 n = normalize(normal);
    vec3 dirLight0 = normalize(vec3(1,1,1));
    vec3 s = normalize(normalMatrix * dirLight0);

    float k_d = 0.7;
    float i_d = k_d * max(0, dot(s, n));
    float i_a = 0.2;
    vec3 baseColor = vec3(1,1,1);

    float i_total = i_d + i_a;

    gl_FragColor = vec4(i_total*baseColor, 1);
}
```

To improve the visual quality of produced terrain, and give a better impression of how terrain would look like, an automatic texturing shader was also implemented. This shader was implemented as a fragment shader, using implicit procedural texturing techniques, hard-coded in GLSL. The approach for the shader is rather simple:

1. A grass texture is computed using fBm.
2. The grass texture is then blended with a rock color according to terrain incline, in order to make the grass disappear from too steep slopes.
3. A "snow line" is computed using three-octave fBm in order to determine the height at which snow should appear.
4. The grass-and-rock texture is then blended with bright white for snow according to the snowline.

5. A similar “beach line” is computed and a sand color blended in for all lower heights.
6. A blue color is blended in for all heights below zero to emulate water.
7. The surface normal is perturbed using fBm to make the single colored texture appear less uniform.

The application lets the user switch between the two fragment shaders by clicking a checkbox. In Fig. 4.4, a side-by-side comparison between the two texturing modes is shown.

Handling terrain changes

The terrain height function itself is dynamically generated by calling `nm::glsl::GlslGenerator::co`
The function takes the input and output of a terrain user module as arguments and returns a string containing GLSL source code. This source code is then concatenated with the vertex shader code, to make sure it is callable from the vertex shader itself. Each time changes are made to the terrain generator, new source code for the height function is generated and the shader program is automatically recompiled.

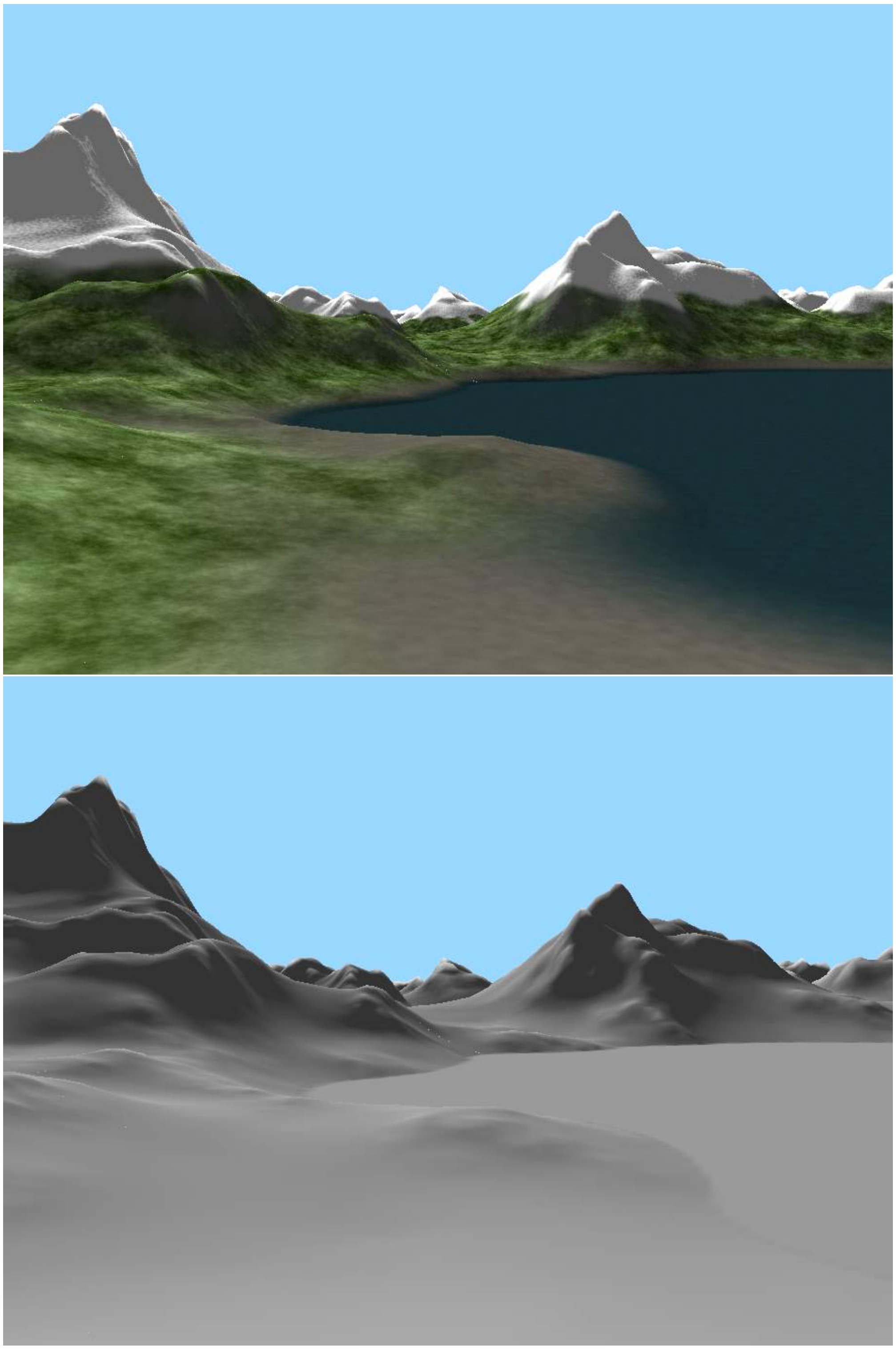


Figure 4.4: Side-by-side comparison of the terrain preview using different fragment shaders.

Chapter 5

Results and Discussion

In this chapter, our results will be presented and discussed. The performance of the system will be benchmarked against CPU-libraries with similar features. The run-time complexity of generated GLSL code will be discussed and compared against other libraries.

The feature set of our framework will be analyzed and compared against both noise libraries, and offline terrain generators. Advantages and limitations of our representation for stochastic implicit terrain surfaces will also be discussed and compared against noise libraries.

While there was not enough time to perform a large user test, the general response from our pilot tests will be summarized, and some content created by the testers will be showcased. The usability of the system will be discussed, and suggestions for improvements given.

Finally, the software qualities of the developed framework will be discussed. Of particular interest is whether the implementation meets the non-functional requirements for modifiability. It will be explained how the framework could be extended to support voxel terrains and evaluation on additional platforms.

5.1 Benchmarking

In order to assess whether heightmap generation would be fast enough for real-time editing, a benchmark terrain was created for the framework. A similar benchmark was also created for ANL and libnoise. The benchmark tested generation of a relatively simple model for a heightmap terrain. The absolute value of 8 octave fractional Brownian motion was clamped between 0.5 and a scaled version of the fBm. This model creates the kind of coastal cliff landscape that can be seen in Fig. 5.1. The code for the benchmarks are included in Appendix C.

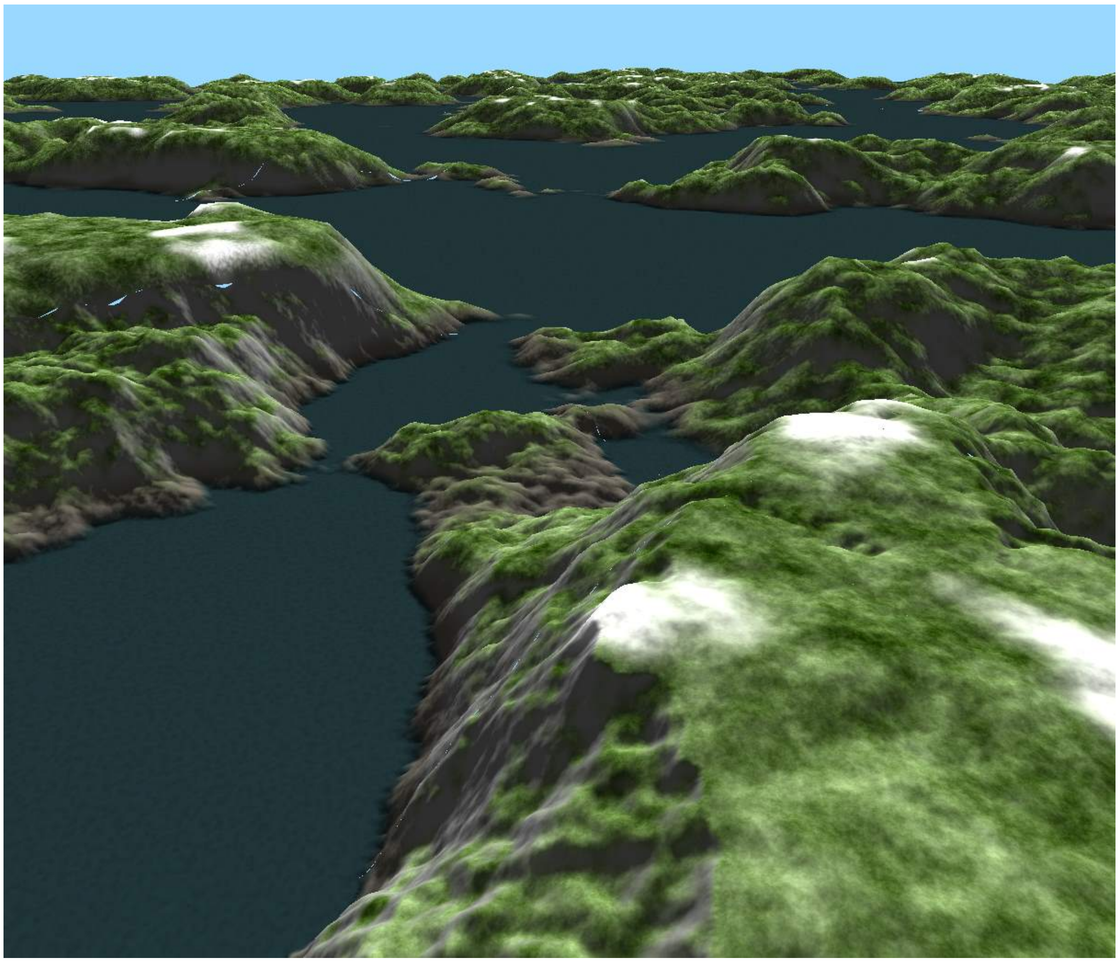


Figure 5.1: The benchmark terrain, rendered by the real-time preview in Noise Modeler.

$$o = 8 \tag{5.1}$$

$$a = fBm(x, y, o) \tag{5.2}$$

$$result = clamp(a, 0.25, 0.75 + 0.25 \cdot a) \tag{5.3}$$

where x and y compose the heightmap position, o is the number of octaves for the fBm function. For simplicity, other parameters of the fBm function have been hidden (lacunarity, gain, etc.), as they do not have a great impact on performance. *clamp* is a common function in graphics that is usually defined as $clamp(a, x, y) = \min(x, \max(y, a))$. The graph representation for this terrain function can be seen in Fig. 5.2.

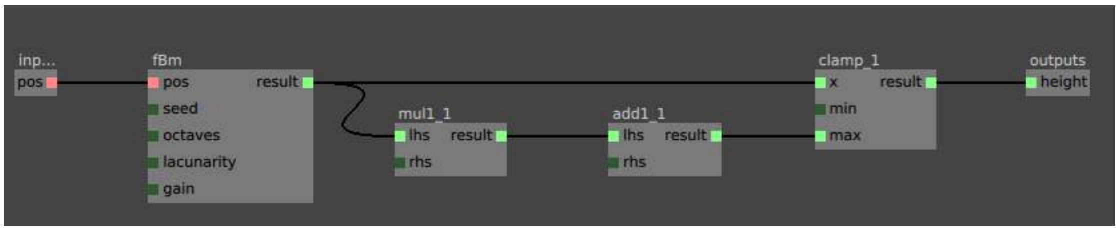


Figure 5.2: The benchmark terrain function, as shown in Noise Modeler.

Due to its age, libnoise does not contain an implementation of simplex noise. Instead, Perlin noise had to be used, which is slightly more expensive for three-dimensional noise.

This test scenario may not be a completely fair comparison: libnoise is optimized for three-dimensional noise, and it might be considered unfair to compare CPU implementations against a GPU implementation. However, these are the libraries that are commonly used to implement noise-based terrains. So although it is possible to hand-write a GPU implementation that performs similarly to our generated GLSL code, it is not commonly carried out in practice, and there is little middleware available to make this process easier. Hence, our benchmark might be considered fair because it benchmarks Noise Modeler against other tools that game engine developers commonly choose for the same task.

During the benchmark, we generated 8192^2 points, as this corresponds to the maximum allowed heightmap size in Unreal Engine [11]. For all benchmarks, the heightmap is first generated and then stored in the host memory. Consequently, the texture has to be transferred from the GPU to the CPU in the Noise Modeler benchmark. All tests and libraries were compiled with gcc 4.9.0 using the "-O3" optimization option, as recommended by the libnoise documentation [54]. libnoise 1.0.0 and ANL revision dea64e1c14d0 (8 November 2013) were used.

Table 5.1 shows the results of our benchmark, using both libnoise and ANL with, and without caching for the fBm module. Note that there is only one column for our framework, because caching is implicit in the model (see Section 5.3). There are, however, three different results for the Noise Modeler. The first result is the total time of the benchmark, including OpenGL context creation, reading and parsing the terrain file from disk, compiling shader source, allocating GPU memory, setting OpenGL state, executing the shader program, and transferring the results back to CPU memory. Context creation is a rather expensive operation, and is typically only done once when an application is started. Hence, the second result includes only the time from starting the shader program until the results were back at the CPU which gives an indication of how long it takes to generate the terrain once a program has been started. The third and final result does not include a transfer back to the host memory.

	libnoise	libnoise, cached	ANL	ANL, cached	Noise Modeler
PC 1	153 735 ms	79 402 ms	102 640 ms	62 370 ms	973/558/243 ms
PC 2	111 350 ms	55 880 ms	67 200 ms	39 470 ms	485/355/268 ms*
PC 3	190 890 ms	99 060 ms	113 180 ms	66 670 ms	969/650/192 ms

Table 5.1: Benchmark results. Generating a 8192×8192 patch of heightmap terrain.

* The results are for a 4096×4096 texture, because Intel HD 4000 does not support 8192×8192 framebuffers.

	PC 1	PC 2	PC 3
GPU	AMD Radeon 4870 HD	Intel HD 4000	NVIDIA GeForce 460 GTX
GPU Driver	Mesa 10.1.4	Mesa 10.1.4	NVIDIA 337.25
CPU	AMD Phenom II X3 720	Intel Core i5-3317U @ 1.7 GHz	Intel Core i7 930 @ 2.8 GHz
Memory	4 GB	4 GB	10 GB
Build year	2008	2012	

Table 5.2: Benchmark configurations. All machines ran Arch Linux (updated 28 May 2014) with kernel version 3.14.4.

Table 5.2 shows the configuration of the test machines. Notable here is that PC 1 used the open-source Radeon drivers, which are significantly slower than the proprietary drivers[1].

Note that this benchmark tests how long it takes to generate a complete heightmap of $8192^2 = 67\,108\,864$ points. However, as explained in Section 3.3.1, it may not be necessary to generate a complete heightmap. In many rendering implementations, such as the preview in Noise Modeler, it is not necessary to generate a complete heightmap, points can merely be generated when they are needed. In the initial position, the preview in Noise Modeler renders 88 grids of resolution 64×64. This means that only $88 \cdot 64 \cdot 64 = 360\,448$ height values are needed. If a more sophisticated tessellation algorithm is deployed, the number of needed heights may only be a fraction of this.

If the results in Table 5.1 for PC 1 are generalized, it may give an estimate for the constraints the libraries impose for the lower limit for time per frame in the preview. The time needed to render a preview t_p, may be approximated using the following equation:

$$t_p \approx t_b \frac{v_p}{v_b}$$

where t_b is the total benchmark time, v_b is the number of vertices in the benchmark and v_p is the number of vertices needed to render a preview.

[1]A recent benchmark of the drivers can be found here: `http://www.phoronix.com/scan.php?page=article&item=radeon_1404_win81`

Using this formula yields the following minimum frame times for the tested noise libraries:

libnoise: 420 ms

ANL: 330 ms

Noise Modeler: 1.3 ms

If we reduce the quality of the preview, it might seem like even the CPU implementations could almost be fast enough for real time generation. However, due to the implementations running on the host, rendering a terrain still requires expensive memory transfers between the host and the GPU. The solution would also scale poorly as the complexity of the terrain function increases.

Furthermore, it is impractical to combine CPU implementations with hardware tessellation, because tessellation would happen as part of the rendering call, after the terrain has been generated. Our approach, on the other hand, is perfectly suited to take advantage of hardware tessellation, because evaluation may take place at any stage in the rendering pipeline.

In Section 3.3.1, it was suggested that it must be possible to generate enough terrain for a preview in under 300 ms, and, ideally, under 50 ms, in order to use it for real-time editing. Although our tests show only 1.3 ms as the time required to generate enough terrain, this is a lower limit. More complex terrains will also involve correspondingly more expensive computations, so it is desirable to be able to tolerate an increase in computation time. Since the benchmark indicates that generation is over 200 times faster than required (see Section 3.3.1) for a simple model, it might be considered reasonable to suppose that it will be fast enough for a more complex model as well.

Both the benchmarks, and the responses from the pilot tests suggest that requirement Perf1 has clearly been fulfilled, even if the model is several times more complex than in our benchmark.

5.2 Implemented and missing features

Below the implemented and missing features of Noise Modeler are discussed and compared with the feature sets of related frameworks and libraries.

5.2.1 Real-time preview

A real-time preview was implemented. On a four year old graphics card (see PC 3 in Section 5.1) a terrain preview with more than 350 000 vertices was drawn at an average frame rate of 60 frames per second. Higher frame rates might also have been achieved if the application had not been limited by the Qt framework's vertical sync[2].

When a change to the module graph occurs, the GLSL shader code has to be regenerated and recompiled. This is handled automatically by the framework. If the graph is changed every frame, the frame rate drops to around 15 frames per second, or 66 ms per frame. While the user may perceive the drop in frame rate, the *input* delay is barely noticeable, which means editing operations will still be felt as instantaneous, but the illusion of motion may be lost. On newer hardware, this problem will be less significant.

On the other hand, if only the inputs to the generated GLSL shader are changed, there is no need for recompilation, and the frame rate can stay at a stable 60 frames per second. The speed of the generation is benchmarked further in Section 5.1.

The texturing of the preview is very limited. The user only has two choices, diffuse white shading, or our improvised automatic height-based shading. A more dynamic approach would have been desirable, letting the user influence the texturing. This could perhaps be done by allowing the user to create procedural alpha maps and use a texture splatting shader [9].

While the implemented terrain texturing may be simplistic and inflexible, it still shows the shape of the terrain clearly, solving its primary objective: providing immediate and informative user feedback.

5.2.2 Generator functions

libnoise and ANL both have a set of modules they call "generator functions". These are typically functions that create fractal noise, or terrains without depending on other modules. Table 5.3 shows support for different functions in the three libraries:

This is not an exhaustive list, however; ANL also includes several more exotic algorithms that have not been included here. ANL clearly has the largest amount of available algorithms, and while libnoise has pretty wide support, it is limited to only the three-dimensional variants.

Our framework does not offer billow noise, Voronoi patterns and Worley noise. The reason for this is simply lack of development time. Porting difficulties of these algorithms have not been evaluated.

Aside from the algorithms listed here, these libraries commonly support a wide range of other generation functions such as checkered patterns, circles, spotted

[2]Vertical sync is a feature that synchronizes the swapping of frame buffers with the update frequency of display. This is done to prevent rendering frames that will never be displayed.

Algorithm	ANL	libnoise	nmlib
fBm	Yes	Yes	Yes
Ridged multifractal	Yes	Yes	Yes
Hybrid multifractal	Yes	No	Yes
Billow noise	Yes	Yes	No
Voronoi	Yes	Yes	No
Worley noise	Yes	No	No

Table 5.3: Support for various common generation algorithms in different frameworks.

patterns and similar. None of these are supported directly by our framework, but they can often be built from more basic elements and be encapsulated as a new user type that can be used similarly. To construct a checkered pattern, for instance, it is possible to combine “mod” and “step” modules. To keep the library simple, however, we chose not to implement generator functions that could easily be expressed by other functions, as this would make the representation unnecessarily complicated and harder to port to new platforms.

5.2.3 Lack of erosion algorithms

Erosion algorithms are a popular feature in many offline terrain generation tools, and their absence will surely be noticed by users of these applications. Their appeal is that they are conceptually easy to understand, and that they produce impressive results compared to the amount of work required by the user.

The erosion models described in Section 2.4 are designed to run on heightmaps to produce a new eroded heightmap. This means an original complete heightmap has to be provided as input for the algorithm. These algorithms do not fit the paradigm of functional composition for stochastic surfaces, where each point is independent from its neighbors. Consequently, heightmap erosion algorithms can not be expressed as module types in our framework.

Other frameworks that use implicit surfaces to model terrains, *and* support erosion algorithms frequently achieve this by introducing a concept of “post-effects”. These post-effects are algorithms that can be run on a terrain once it has been expanded from the height function into a heightmap [36], [61].

Although our framework does not support erosion algorithms, the game engine developer is still free to use our framework to create a preliminary heightmap, and then use an external erosion tool to improve the quality of the heightmap. While this makes it possible to integrate with erosion tools, it would mean the previews in the GUI application would no longer reflect the final terrain accurately.

Note that erosion algorithms are also problematic to use on *endless* terrains in general, because most of them assume a heightmap with finite dimensions. Running the algorithm on adjacent patches of a heightmap terrain separately will cause discontinuities in the terrain height values, resulting in visible stitches.

5.3 Run-time complexity of generated GLSL code

In this section we will look at how the run-time complexity of generated GLSL code changes with different module graphs.

When the graph model is used to generate GLSL code, a piece of inline code is generated for each module that is needed by the computation. Graphs in the model that are not needed to produce the requested outputs are omitted by the code generation tool. Generated code for each needed module is concatenated and will be computed in sequence on the GPU. If we assume a constant run-time per module, this means that the worst-case complexity of the generated code is bounded from above by:

$$O\left(\sum_{m \in G} T(1)\right) = O(MT(1)) = O(M) \tag{5.4}$$

where m is a module in the module graph, G, and M is the total number of moudules in G.

Note that this is only a worst-case run-time. Since the code is generated inline, and then handed to the GLSL compiler, many optimizations can be performed by the compiler. This means compiler optimizations such as

- dead code elimination
- copy propagation
- loop unrolling
- code motion
- function inlining
- constant folding
- constant propagation
- reduction in strength

are applied to the resulting code [62].

For example: Assume one of the inputs to a multiplication module is specified as a constant, 2, while the other is bound to the output of another module. The code generation tool will generate the following GLSL code:

Listing 5.1: Generated code for a multiplication module, stripped of comments.

```
 1  float nm_id_4;
 2  {
 3      float lhs;
 4      float rhs;
 5      lhs = 1;
 6      rhs = 2;
 7      lhs = nm_id_3;
 8      float result = lhs * rhs;
 9      nm_id_4 = result;
10  }
```

The compiler will then perform the following optimizations:

- Dead code elimination will remove the assignment in line 5.
- Constant propagation will replace the "rhs" variable with the constant, 2.
- Reduction of strength will be applied to line 8 which now contains the expression `lhs*2`, replacing it with an addition, `lhs+lhs`, or a left shift, `lhs<<1`.
- Copy propagation is applied to line 9.

In this example, it has been shown how code may often be optimized when inputs of a module are constant. Although this showed how optimization may happen within one module, the compiler will also perform optimizations across modules, perhaps even eliminating the execution of whole modules.

In ANL and libnoise, evaluation of module graphs are happening directly in the graph model, without generating specialized code that may be optimized by a compiler. This means that for our example, with a multiplication module having a constant 2 as one of its inputs, ANL or libnoise would use the multiplication operation instead of the more inexpensive addition or shift operations.

Due to the differences in the conceptual model, our approach has an additional advantage over ANL and libnoise performance-wise. Earlier, it was shown that the run time complexity of our approach was $O(M)$. In ANL and libnoise, however, a shared source module does not automatically mean shared computation. On the contrary, since the computation of a module's output is repeated each time it is used as a source, the algorithm corresponds to a depth first traversal of a DAG where nodes are not marked when visited.

As discussed earlier, this flaw is handled in ANL and libnoise by introducing cache modules that store the result of the last computation.

There are two problems with this approach: Firstly, it requires a user that is aware of the problem and knows when it is appropriate to insert a cache module. Secondly, a cache module may easily be defeated if the user inserts a turbulence module, which means that the position in each function call may be different from the last, thus thrashing the cache.

If caching modules are used extensively, and position-changing modules are avoided, the libnoise approach can achieve the same $O(M)$ complexity as our approach.

In our approach, however, the position is explicit in the graph. This means that it is not possible to use turbulence modules in that manner. With the libnoise approach, care has to be taken to avoid expensive recalculations, while in our approach, it is implicit in the model.

5.4 Pilot testing

While one of the main motivations for this thesis was to enable a more intuitive terrain design process, usability been a priority of the thesis work. The thesis work has been more concerned with developing the generation engine required to create an intuitive interface.

Although usability were not in focus, pilot testing still provided some useful feedback regarding the user interface. As explained in Section 3.6, the main tester was a game developer potentially interested in using the framework for one of his projects[3]. Other testers were not involved in game development and were recruited among friends and relatives of the developer. A summary of the received feedback given will be presented below.

There proved to be much room for improvement in the user interface. Testers often needed help to discover basic functionality, such as how to navigate the preview window. For the non-developer testers there also seemed to be general confusion as to what the purpose of the program was. These testers struggled to understand what the graph interface was representing and how a height function could represent a terrain.

The game developer, however, picked up the concept quickly, as he had some prior familiarity with concepts such as heightmaps and noise functions. He described the framework as “for the most part intuitive”, and was able to use the program without any assistance. Fig. 5.3 shows a terrain designed by him.

Our limited amount of testing suggests that non-developers would need a significant amount of training in order to use the program efficiently, while developers

[3]His project can be found here: `https://github.com/monkeybits/primordial`

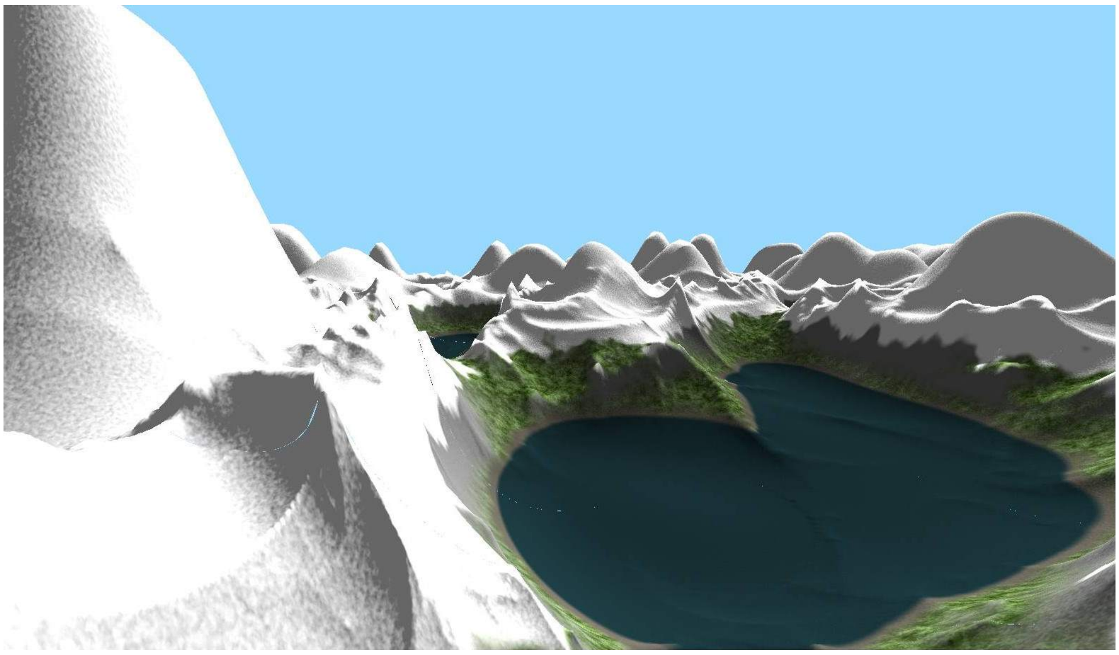


Figure 5.3: A terrain designed by the primary test subject

would be able to use the program without any training at all.

On one hand, this indicates that the user interface is severely lacking in terms of usability. On the other hand, the target audience of the software is indeed game developers, and it may not be completely unreasonable to assume familiarity with common game development concepts such as heightmaps.

The alternative to using our program, is to program against a noise library. Not only does this require the user to know a programming language, but the user must also learn how to program against the library API.

It can be concluded that while our program has room for improvement, it is still a significant improvement over programming against a noise library.

5.5 Software quality

As planned in the framework architecture, the developed software is highly modular. It is designed so that parts of the system can easily be swapped out.

Most importantly, there is a clear distinction between the GUI application and the library. This allows the GUI to be replaced or removed, and GLSL code generation can be executed without involving the GUI and its dependencies. This has been demonstrated by creating a benchmark application that evaluates a terrain on the GPU and loads it into CPU memory, ready to use. Another demonstration of this modularity is the creation of a simple command-line program for generating GLSL shaders from “nm.json”-documents.

The system is also modular on the library level, meaning that there are clear boundaries between the “model”, “serialization”, and “code generation” software modules. Either or both of the serialization and code generation modules may be removed without affecting the rest of the library.

5.5.1 Supporting new evaluation platforms

Extending the framework with code generation for new languages may be done by creating a new library module that traverses the model and generates code for that platform.

For languages similar to C, another option is available: Most of the code within the code generation software module is not GLSL specific. Hence, support for a new C-like language may be added simply by sub-classing the `nm::InlineGenerator` class and overloading the **`genTypeKeyword`**, **`genDeclaration`**, **`genAssignment`**, **`genVariable`**, **`genValue`** and **`genFunctionCall`** methods as appropriate.

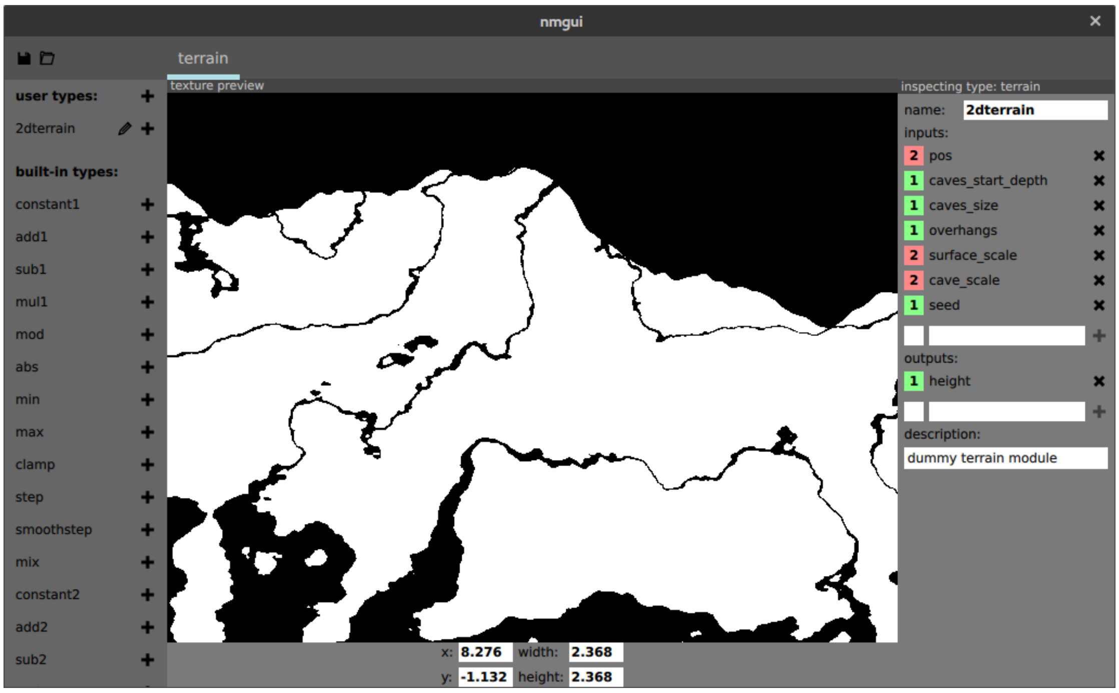
Code for each built-in module type must also be implemented in the new language. This can be done by following the definition for the module types in Table A.1. Special care has to be taken when implementing “noise”, “fBm”, “ridgedmultifractal” and “hybridmultifractal”, however, to make sure that they use the exact same approach as our GLSL implementations, or the terrain will not be deterministic across platforms.

5.5.2 Supporting additional terrain types

All the code in the library, and almost all code in the GUI application, has been written without being tailored for heightmap terrains. The only remaining obstacle for supporting voxel terrains and vector displacement terrains is the development of preview rendering code for these terrain types. By following the rendering approach of Geiss [17], it should be straightforward to insert a density function generated by our existing GLSL code generation tools.

To reiterate more clearly: The library already supports multiple types of terrain, and the GUI application is only lacking previewing functionality.

In fact, the GUI application is already capable of modeling and previewing terrains for two-dimensional side-scrolling games (see Fig. 5.4).



name: 2dterrain
inputs:
2 pos
1 caves_start_depth
1 caves_size
1 overhangs
2 surface_scale
2 cave_scale
1 seed
outputs:
1 height
description:
dummy terrain module

x: 8.276 width: 2.368
y: -1.132 height: 2.368

Figure 5.4: A two-dimensional terrain with an underground cave system designed with Noise Modeler. The terrain is seen from the side, with the ground in white and the air and caves in black. This terrain may be used by side-scrolling games such as *Terraria*, *Starbound* or *Worms*.

5.6 Utility as a game development tool

While shader editors use a similar approach for generating shaders through a flow-graph interface, they are aimed at textures and do not support terrain previews. Such tools could be used to create similar terrain models, but the designer would then create blindly, only aided by a texture representation of the terrain.

Offline procedural terrain editors often also feature a flow-graph interface for constructing a fully procedural terrain. They do not, however, have the ability to generate terrains in-game, or generate shader code for terrain functions.

ANL, libnoise, and GeoGen are all designed with online terrain generation in mind. However, they all require the designer to work in a scripting or programming language. Consequently, previewing results often requires recompilation and may cause significant delays since the libraries are implemented on the CPU.

To our knowledge, the Noise Modeler framework is therefore unique in its cause. It may be limited in features, and rough around the edges in terms of usability, but it clearly outperforms existing noise libraries, and offers terrain specific features and heightmap previews not present in procedural shader editors.

Since the framework is released as open-source, under a permissive license, we believe it should be considered as attractive middleware for game developers looking to implement endless procedural worlds. Fixing its flaws and implement its missing features is probably worth the trouble instead of continuing the current practice of programming blindly against noise APIs.

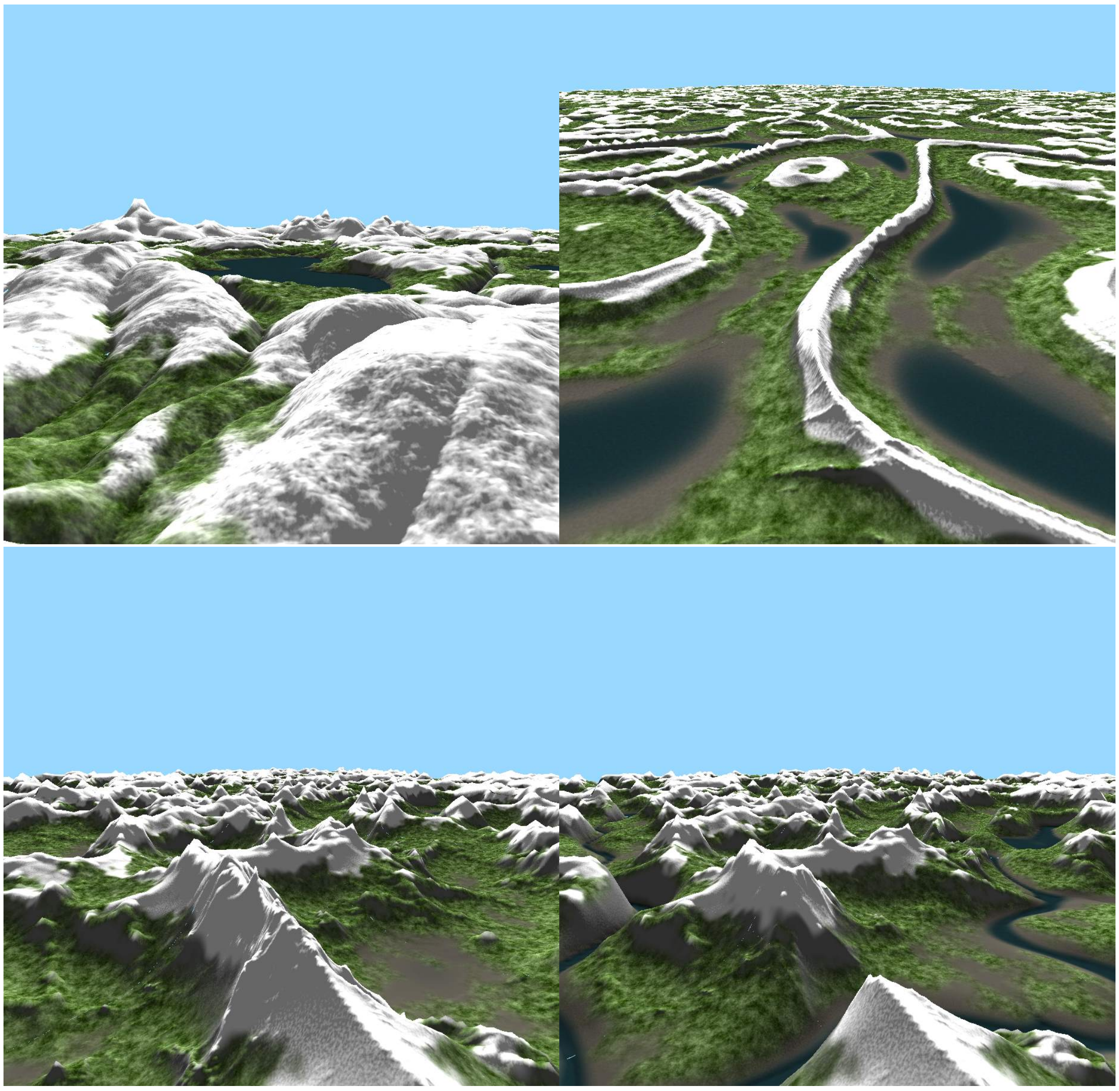


Figure 5.5: A few terrains designed with Noise Modeler. The bottom row shows how user types may be used to combine a mountain terrain and a river network.

Chapter 6

Conclusions and Future Work

Generating endless terrains online using the GPU is not a new idea. Neither is using the GPU to augment terrain editors with responsive procedural tools. The novelty of this thesis lies not in the individual techniques used, but from the effort to unify them. Most terrain editors focus on offline procedural terrains, ignoring the powerful capabilities of procedural generation, the most important being replayability and vastness. The Noise Modeler framework shows that 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.

A novel method for modeling stochastic implicit terrain surfaces has been proposed, and a matching serialization format has been designed. The proposed solution has been tested by implementing a proof-of-concept framework consisting of a generation library and a GUI application for editing and previewing terrains in real-time.

The developed GUI application has shown that the modeling approach may successfully be used to model heightmap terrains, supporting a combination of popular algorithms for stochastic implicit terrains. Furthermore, it has been established that our framework is capable of modeling and generating — although not previewing — other types of terrains, including voxel terrains and vector displacement terrains. This indicates that our terrain modeling approach is flexible and may integrate well with a variety of game engines with different approaches to terrain modeling.

By computing terrain geometry using the GPU, it has been demonstrated that our representation is a good fit for massively parallel architectures. It has also been pointed out how the approach is expected to feature a near linear speedup with all problem sizes except the very smallest. The efficient computation of height values allowed a high-quality terrain preview to be updated in real time with a vertex count comparable to state-of-the-art video games.

A high-quality preview with interactive performance has made it possible to develop a unique tool. It has been shown how the tool allows user-friendly editing with real-time, continuous input. We have also explained why the framework as a whole may be attractive to game developers due to its unique combination of features, namely featuring both an editing application with immediate high-quality feedback, *and* the ability to efficiently generate terrains as a library. Furthermore, our careful design of the software architecture has resulted in a generation library with almost no dependencies and a software license compatible with commercial game development.

The thesis work has shown that by representing stochastic implicit terrains as a graph of modules, it is possible to generate, compile and execute a GLSL shader program quickly enough to enable interactive editing of terrains with immediate visual feedback.

6.1 Future work

In this section, we offer various suggestions of how the thesis work may be continued.

6.1.1 Improvement of heightmap rendering

As explained in Section 2.10, the terrain is rendered using improvised procedural texturing that is not very configurable. This is a part of the framework that has a huge room for improvement. By implementing support for textured terrains, normal mapping and dynamic lighting, the visual quality of the rendered terrains could be greatly improved. If more effort was put into the environment, such as by adding a skybox[1], fog, or a dynamic day-and night cycle, it would also be easier to get an impression of what the terrain would look like inside a game engine.

6.1.2 Platform support

There is currently only support for GLSL code generation. The code generation module is built to be extensible, and adding support for similar languages such as C, HLSL and OpenCL and CUDA should be straightforward.

Another useful feature would be to support evaluation points directly, without the intermediate code generation step. This would allow easier integration with non-rendering frameworks such as path-finding and AI. These systems are typically

[1]A skybox is a method of drawing parts of the world that are far away and unreachable by the player using simple geometry (usually the inside of a cube) at a fixed distance from the observer.

implemented on the CPU, and supporting direct queries to the model would make this process much easier.

6.1.3 Other terrain paradigms

The GUI application is currently specialized for heightmap terrains. This does not need to be the case. The library is already capable of generating models for voxel terrains, but there is currently no support for previewing such terrains in the GUI application. By extending the application to include voxel terrain previews, and implementing additional three-dimensional module types for the library, the framework could become a powerful tool for games with voxel terrains.

It would also be interesting to extend the heightmap preview to support vector displacement terrains as well. This is particularly interesting because very few tools exist for this type of terrain representation.

6.1.4 Integration with existing frameworks

In order to make the framework usable for game developers, the framework should be as easy as possible to integrate with their game engine. Many game engines have a plug-in architecture that could be taken advantage of in order to make integration with these engines as seamless as possible.

In the current state of Noise Modeler, the programmer has to do a significant amount of work in order to make our approach compatible with their game engine. See the benchmark code in Appendix C for an idea of how much work is required to get a patch of generated terrain into CPU memory. More of this process could be integrated as part of the framework, or as ready-to-use plug-ins.

6.1.5 More advanced built-in modules

As explained in Section 5.2, Noise Modeler lacks some features, that are often expected to be available in noise libraries. Although billow noise, Voronoi patterns, and Worley noise is are not available, they should be relatively straightforward to implement in GLSL. A natural next step for the project, is to implement these missing algorithms for 1 to 4 dimensions.

The most glaringly missing feature of the framework, is support for thermal and hydraulic erosion algorithms. Unfortunately, most erosion algorithms operate by iteratively modifying a bounded grid of a fixed resolution. This approach is not compatible with our approach for code generation and single point evaluation. Note that it is still possible to run erosion algorithms *after* a heightmap has been generated, as a post-processing step. However, this can not be included as part of the module graph. A great topic for another research project would be to investigate whether it is possible to develop new erosion algorithms that will run efficiently for point evaluation algorithms.

6.1.6 Combine with generate-and-test algorithms

It would be interesting to find out whether our approach could be combined with search-based procedural techniques to fully automate terrain creation. Different parts of the graph might perhaps be used with a genetic algorithm to randomize the structure of the module graph. A utility mechanism would then have to be developed to guide the search. This could either be done manually, or algorithmically.

Bibliography

[1] K. Perlin, “An image synthesizer”, *ACM SIGGRAPH Computer Graphics*, vol. 19, no. 3, pp. 287–296, 1985.

[2] F. K. Musgrave, C. E. Kolb, and R. S. Mace, “The synthesis and rendering of eroded fractal terrains”, in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 23, 1989, pp. 41–50.

[3] F. K. Musgrave, “Procedural fractal terrains”, in *Texturing & Modeling: A Procedural Approach*, Morgan Kaufmann, 2003.

[4] M. Gamito, “Techniques for stochastic implicit surface modelling and rendering”, PhD thesis, University of Sheffield, England, 2009.

[5] World Machine Software. (2014). World machine 2 user’s manual, [Online]. Available: http://www.world-machine.com/learn.php?page=userguide (visited on 05/21/2014).

[6] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, “Search-based procedural content generation: a taxonomy and survey”, *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 3, no. 3, pp. 172–186, 2011.

[7] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, “Declarative terrain modeling for military training games”, *International Journal of Computer Games Technology*, vol. 2010, p. 2, 2010.

[8] N. Shaker, J. Togelius, and M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2014.

[9] J. Andersson, “Terrain rendering in Frostbite using procedural shader splatting”, in *ACM SIGGRAPH 2007 courses*, ACM, 2007, pp. 38–58.

[10] M. Gollent, “Landscape creation and rendering in REDengine 3”, Game Developers Conference 2014, 2014, [Online]. Available: http://twvideo01.ubm-us.net/o1/vault/GDC2014/Presentations/Gollent_Marcin_Landscape_Creation_and.pdf (visited on 07/09/2014).

[11] Epic Games. (2014). Unreal Engine 4 landscape outdoor terrain system, [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/Landscape/index.html (visited on 06/03/2014).

[12] M. Widmark, "Terrain in Battlefield 3: A modern complete and scalable system", Game Developers Conference 2012, 2012, [Online]. Available: http://gdcvault.com/play/1015415/Terrain-in-Battlefield-3-A (visited on 06/03/2014).

[13] Unity Technologies. (2014). Unity3D terrain engine guide, [Online]. Available: http://docs.unity3d.com/420/Documentation/Components/script-Terrain.html (visited on 06/03/2014).

[14] C. McAnlis, "Halo Wars: The terrain of next-gen", Game Developers Conference 2009, 2009, [Online]. Available: http://www.gdcvault.com/play/1277/HALO-WARS-The-Terrain-of (visited on 04/28/2014).

[15] B. Benes and R. Forsbach, "Layered data representation for visual simulation of terrain erosion", in *Computer Graphics, Spring Conference on, 2001.*, IEEE, 2001, pp. 80–86.

[16] W. E. Lorensen and H. E. Cline, "Marching cubes: a high resolution 3d surface construction algorithm", in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 21, 1987, pp. 163–169.

[17] R. Geiss, "Generating complex procedural terrains using the GPU", in *GPU Gems 3*, Addison-Wesley Professional, 2007, pp. 7–37.

[18] Crytek. (2014). CryENGINE documentation: static vs. dynamic lighting, [Online]. Available: http://docs.cryengine.com/display/SDKDOC4/Static+vs.+Dynamic+Lighting (visited on 06/04/2014).

[19] B. Mandelbrot, *The Fractal Geometry of Nature*. CA: Freeman, 1982.

[20] A. Fournier, D. Fussell, and L. Carpenter, "Computer rendering of stochastic models", *Communications of the ACM*, vol. 25, no. 6, pp. 371–384, 1982.

[21] G. S. Miller, "The definition and rendering of terrain maps", in *ACM SIGGRAPH Computer Graphics*, ACM, vol. 20, 1986, pp. 39–48.

[22] M. N. Gamito and F. K. Musgrave, "Procedural landscapes with overhangs", in *10th Portuguese Computer Graphics Meeting*, vol. 2, 2001.

[23] A. Lagae, S. Lefebvre, R. Cook, T. Derose, G. Drettakis, D. S. Ebert, J. Lewis, K. Perlin, and M. Zwicker, "State of the art in procedural noise functions", *Eurographics 2010-State of the Art Reports*, 2010.

[24] M. Olano, J. Hart, W Heidrich, B Mark, and K Perlin, "Real-time shading languages", *Course Notes. ACM SIGGRAPH*, 2002.

[25] K. Perlin, "Improving noise", in *ACM Transactions on Graphics (TOG)*, ACM, vol. 21, 2002, pp. 681–682.

[26] ——, "Implementing improved Perlin noise", in *GPU Gems*, Addison-Wesley, 2004, pp. 73–85.

[27] K. Babington, "Terrain rendering techniques for the HPC-lab snow simulator", Master's Thesis, Norwegian University of Science and Technology, 2012.

[28] A. 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.

[29] R. M. Smelik, T. Tutenel, R. Bidarra, and B. Benes, "A survey on procedural modelling for virtual worlds", in *Computer Graphics Forum*, Wiley Online Library, 2014.

[30] J. Olsen, "Realtime procedural terrain generation", University of Southern Denmark, Tech. Rep., 2004.

[31] X. Mei, P. Decaudin, and B.-G. Hu, "Fast hydraulic erosion simulation and visualization on GPU", in *Computer Graphics and Applications, 2007. PG'07. 15th Pacific Conference on*, IEEE, 2007, pp. 47–56.

[32] P. Krištof, B. Beneš, J Křivánek, and O. Št'ava, "Hydraulic erosion using smoothed particle hydrodynamics", in *Computer Graphics Forum*, Wiley Online Library, vol. 28, 2009, pp. 219–228.

[33] D. S. Ebert, F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley, *Texturing & Modeling: A Procedural Approach.* Morgan Kaufmann, 2003.

[34] B. C. Davis, "Terrain generation engine using voxels", Master's thesis, California State University, 2013.

[35] Guruware. (2013). Terranoise, [Online]. Available: `http://www.guruware.at/main/terraNoise/` (visited on 05/21/2014).

[36] J. Tippetts, "Creator of worlds: Procedural terrain generation in a sandbox environment", *Game Developer Magazine*, vol. 18, pp. 20–27, 2011.

[37] F. Bösch. (2014). Lithosphere, [Online]. Available: `http://lithosphere.codeflow.org/` (visited on 03/28/2014).

[38] M. Zábský, *Geogen — Scriptable generator of terrain height maps*, Bachelor thesis. Charles University in Prague, 2011.

[39] Valve Software. (2014). Source Shader Editor, [Online]. Available: `https://developer.valvesoftware.com/wiki/Category:SourceShaderEditor` (visited on 03/28/2014).

[40] J. Busby, Z. Parrish, and J. Wilson, *Mastering Unreal Technology, Volume I: Introduction to Level Design with Unreal Engine 3*. Pearson Education, 2009, vol. 1.

[41] Allegorithmic. (2014). Substance Designer 4, [Online]. Available: http://www.allegorithmic.com/products/substance-designer#features (visited on 03/28/2014).

[42] S. Marison, J. Duplessis, M. Agsen, and T. Pagan, *Visual shader designer*, US Patent App. 13/227,498, 2013. [Online]. Available: http://www.google.com/patents/US20130063460 (visited on 04/28/2014).

[43] R. S. Wright, N. Haemel, G. M. Sellers, and B. Lipchak, *OpenGL SuperBible: Comprehensive Tutorial and Reference*. Pearson Education, 2010.

[44] S. Green, "Implementing improved Perlin noise", in *GPU Gems 2*, Addison-Wesley Professional, 2005, pp. 409–416.

[45] I. McEwan, D. Sheets, M. Richardson, and S. Gustavson, "Efficient computational noise in GLSL", *Journal of Graphics Tools*, vol. 16, no. 2, pp. 85–94, 2012.

[46] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time Optimally Adapting Meshes", in *Proceedings of the 8th Conference on Visualization'97*, IEEE Computer Society Press, 1997, pp. 81–88.

[47] F. Losasso and H. Hoppe, "Geometry clipmaps: Terrain rendering using nested regular grids", *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3, pp. 769–776, 2004.

[48] A. Asirvatham and H. Hoppe, "Terrain rendering using GPU-based geometry clipmaps", in *GPU Gems 2*, Addison-Wesley Professional, 2005, pp. 27–46.

[49] F. Strugar, "Continuous distance-dependent level of detail for rendering heightmaps (CDLOD)", *Journal of Graphics, GPU, and Game Tools*, vol. 14, no. 4, pp. 57–74, 2009.

[50] J. Thelin, "Quick user interfaces with Qt", *Linux Journal*, vol. 2011, no. 204, p. 7, 2011.

[51] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[52] R. M. Smelik, T. Tutenel, K. J. de Kraker, and R. Bidarra, "A proposal for a procedural terrain modelling framework", in *Poster Proceedings of the 14th Eurographics Symposium on Virtual Environments EGVE08*, 2008, pp. 39–42.

[53] R. M. Smelik, K. J. De Kraker, T. Tutenel, R. Bidarra, and S. A. Groenewegen, “A survey of procedural methods for terrain modelling”, in *Proceedings of the CASA Workshop on 3D Advanced Media In Gaming And Simulation (3AMIGAS)*, 2009, pp. 25–34.

[54] J. Bevins. (2004). Libnoise documentation, [Online]. Available: `http://libnoise.sourceforge.net/docs/index.html` (visited on 05/20/2014).

[55] J. Tippets. (2011). Accidental Noise Library documentation, [Online]. Available: `http://accidentalnoise.sourceforge.net/docs.html` (visited on 05/20/2014).

[56] J. Rosenberg. (2014). Geocontrol2 features, [Online]. Available: `http://www.geocontrol2.com/e_geocontrol_geocontrol2.htm` (visited on 06/26/2014).

[57] B. Victor, “Inventing on principle”, in *Invited talk at the Canadian University Software Engineering Conference (CUSEC)*, vol. 5, 2012.

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

[59] J. Brooke, “SUS—A quick and dirty usability scale”, *Usability Evaluation in Industry*, vol. 189, p. 194, 1996.

[60] D. Crockford, “The application/json media type for JavaScript Object Notation (JSON)”, IETF, RFC 4627, 2006.

[61] J. K. Helsing, *PTG: Procedural Terrain Generator*, 2012. [Online]. Available: `https://github.com/bobbaluba/PTG` (visited on 06/12/2014).

[62] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2007.

Appendix A

Noise Modeler User's Guide

A.1 Introduction

Noise Modeler is an application for designing noise-based functions for generation of heightmap terrains.

Functions generated may be exported as GLSL, or loaded by an application supporting the "nm.json" file format. If you want to implement support for the file format in your game engine, you may take advantage of the Noise Modeler Library.

A.1.1 Licensing

The source code is distributed under the permissive zlib license. Beware that one of the GUI applications dependencies, Qt, is released under the GPLv3 and LGPL. Consequently, you can not include the GUI in closed-source applications.

However, if you are writing a plug-in for your game or game engine, you will most likely only link against nmlib, which does not depend on Qt. This means you can link statically against nmlib and distribute a closed-source binary of your application/game without changing its licensing.

See the `license.md` file in the root directory of the source code for more information.

A.2 Installation

A.2.1 System requirements

Noise Modeler is designed to be a cross platform application. The application is known to work on Arch Linux and Windows 7. It may also work on OS X and other Unix-based systems.

OpenGL 3.0 support is required in both hardware and software.

A.2.2 Prebuilt binaries

An installer for Windows and an Arch Linux package are available at: `http://www.noisemodeler.org/download.html`.

No other Linux or OS X builds are provided, but the application may be built from source.

A.3 Building from source

This section covers how to build the system on Linux and Windows.

A.3.1 Build dependencies

- git (to get the source code)
- gcc 4.8.1 or newer
- Qt 5.2.1 or newer, including the following modules:
  - QtDeclarative
  - QtSvg
  - QtQuickControls
  - qmake
- Boost.Signals2 (tested with 1_55_0 and newer)
- googletest (only if you are building the unit tests, tested with 1.7.0)

A.3.2 Building on Linux

These instructions are written for Linux, but may apply to other Unix-based systems as well, such as OS X, BSD or Cygwin.

Install build dependencies

Start by installing the build dependencies (Appendix A.3.1).

Arch Linux users may use the following command:

```
$ sudo pacman -S qt5-base qt5-svg qt5-tools \
                 qt5-declarative qt5-quickcontrols \
                 base-devel gtest boost
```

Ubuntu It is easiest if you have version 14.04 or newer, as earlier versions only have outdated development packages in the official repositories. It is possible to get up-to-date versions using PPAs, but this will not be covered here. Install the build dependencies on Ubuntu 14.04 using the following command:

```
$ sudo apt-get install qt-sdk libqt5svg5-dev gcc \
                       libboost-signals-dev libgtest-dev
```

Get and compile the source code

The source code is available on GitHub. To download the source code, enter the following:

```
$ git clone --recursive git@github.com:noisemodeler/noisemodeler↩
   .git
```

It is important to include the `--recursive` option, or you will have to download rapidjson manually.

Then create and enter a build folder where the compilation output will appear:

```
$ mkdir build-noisemodeler
$ cd build-noisemodeler
```

Set up a makefile for your system and Qt version. If you want to build the unit tests as well, append `CONFIG+=build_tests` to the qmake command.

```
$ qmake ../noisemodeler
```

Compile the project:

```
$ make
```

This will produce the following binaries:

build-noisemodeler/nmgui/nmgui The GUI application, "Noise Modeler"

build-noisemodeler/nmlib/nmlib.a A statically compiled version of the library, nmlib.

build-noisemodeler/test_nmlib/test_nmlib Unit tests for nmlib.

build-noisemodeler/test_nmgui/test_nmgui Unit tests for nmgui.

Note that the "nmgui" binary will depend on the Qt shared libraries being installed unless you build with a statically compiled version of Qt.

A.3.3 Building on Windows

Note: You may also install cygwin and attempt to install using the guide in the previous subsection.

Install Qt

Download and install the Qt SDK from `qt-project.org/downloads`. Download the version that says (MinGW, OpenGL). During installation, make sure that you check the MinGW option to install the MinGW toolchain.

Install boost

Download boost and extract it to your harddrive. Add the path to the extracted files to your CPATH environment variable, which tells mingw-gcc where to look for C++ header files.

See `http://www.computerhope.com/issues/ch000549.htm` on how to set environment variables.

You do not need to compile the library, as only header-only libraries are used.

Download the source code

The source code is available on GitHub. To download the source code, enter the following in Git Bash:

```
$ git clone --recursive git@github.com:noisemodeler/noisemodeler↩
    .git
```

Build the project using QtCreator

Or use your favorite graphical git tool.

1. Open "noisemodeler.pro" in the root directory of the project using QtCreator.
2. Click configure project
3. Press Ctrl+R to build and run the GUI application (this may take several minutes).

A.4 Tutorial

In this tutorial we will explain how to use the Noise Modeler application to create a simple coastal landscape.

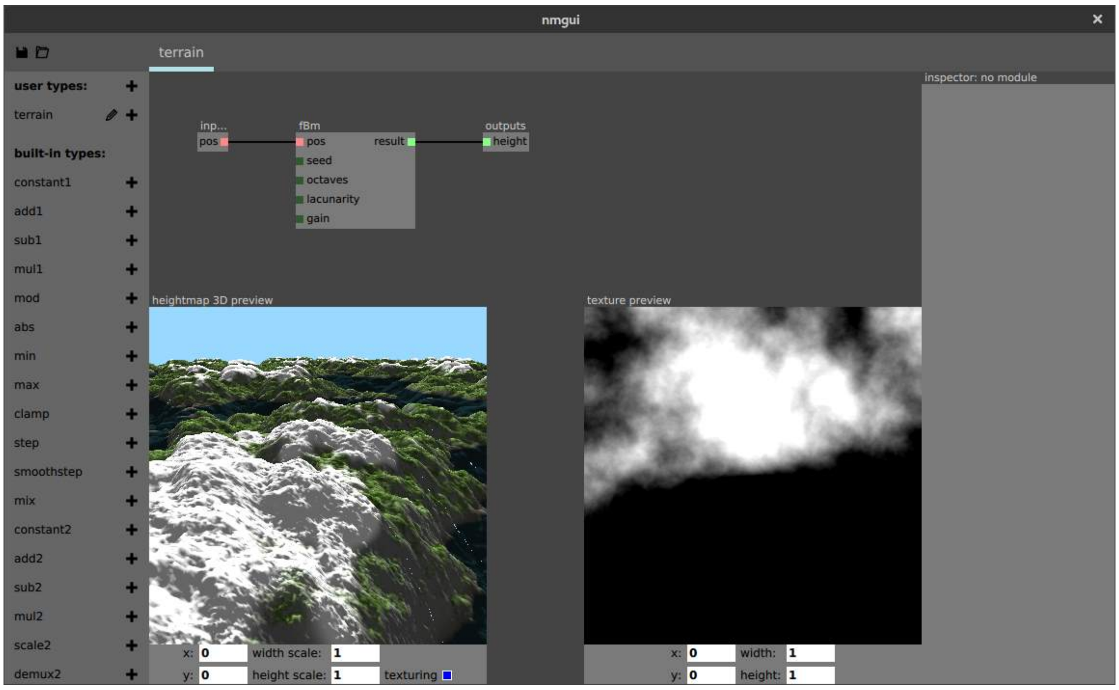


Figure A.1: Screenshot of the application right after it has been started

When you first start the application, you will be greeted with the interface shown in Fig. A.1.

A very simple terrain model has been preloaded, and in the bottom left corner of the application, you will recognize something that looks a bit like a landscape. This preview is your primary feedback tool when editing a terrain, and it is important to know how to navigate it efficiently.

First, double-click the preview. It will now be expanded to a fill the main area of the application. This is very useful when you want to take a closer look at the terrain to verify whether it behaves like intended.

Click and drag inside the 3D preview. You should now see the camera angle changing. You can also use the W, A, S, and D keys to move the camera around. If you have played first-person pc games, this will feel very familiar to you. This way of navigating the terrain is especially useful when exploring the smaller details of the terrain from a ground perspective. If you are more interested in the large-scale features of the terrain, however, this navigation will quickly become impractical if you want to move large distances.

Below the preview, you will find four textboxes, two of them labeled "x" and "y". Entering new values in these fields will move the terrain around. Entering these values using the keyboard, however, is cumbersome. While holding the mouse over one of the text boxes, try scrolling upwards for a while. After a while, you should now see the landcape moving quickly around. All text boxes in the application behave similarly, try scrolling over the text boxes for "width scale" and "height scale" as well, to see their effect on the preview. Now, double click the preview again to un-maximize the preview.

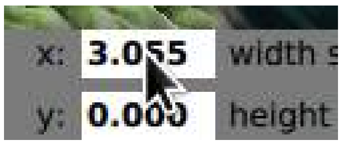


Figure A.2: Use your mouse scroll wheel while hovering over text boxes to easily adjust values.

In the lower right corner, you will see another preview. This preview shows your terrain from above, like a map. White areas of the map correspond to higher values, while black values correspond to low values. This is how a heightmap usually looks when opened with an image-editing application. It is possible to pan and zoom the preview by dragging and scrolling with the mouse, just like in common web applications for maps.

Now that you know how to navigate the previews, let us look at how the terrain can be edited. It might a good idea to restart the application first, so that the configuration of your preview will match ours.

In Noise Modeler, a terrain is modeled by creating a terrain height function. A height function is a function that takes a two-dimensional position argument, and returns the height at that position. This means that the height function can be used to answer questions like: "What is the altitude at this latitude and longtitude?"

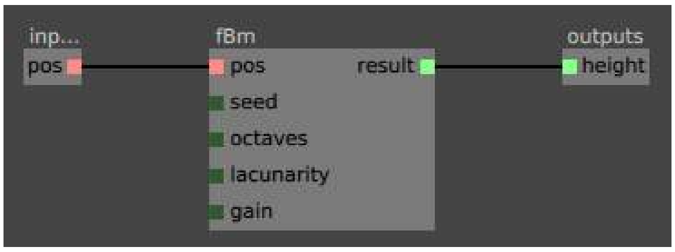


Figure A.3: The default module graph.

In the main area of the application, you will see three boxes labeled “inputs”, “fBm”, and “outputs”. These boxes are called modules, and together they represent a height function. A module is something that transforms inputs into outputs, a function, if you will. Paths between two modules indicate that the output of one module should be the input of another. Values flow from left to right.

The leftmost module, “inputs”, has one output, the position. This value corresponds to one specific length and width position on the terrain. On the other side of the graph, you find the “outputs” module, which has one input, the height. When the terrain preview is generated, different position values enter the graph from the “inputs” module, and some transformations are run on these positions before a final height value reaches the “outputs” module. In our simple graph, the position is only transformed by the “fBm” module.

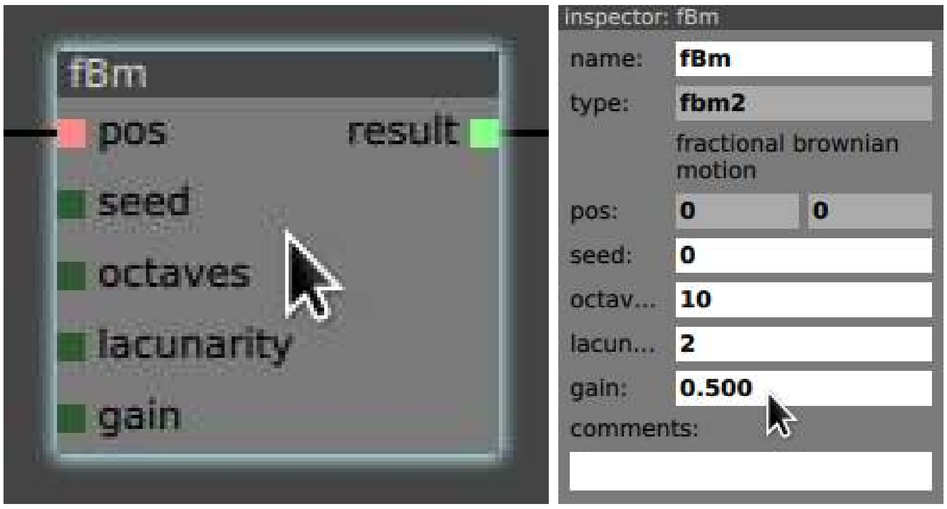
The “fBm” module represents an algorithm called fractional Brownian motion, and constitutes an important building block in the design of procedural fractal terrains. You may perhaps recognize this algorithm from other applications where it might be called “noise”, “fractal noise” or perhaps “advanced Perlin”. We will now guide you through the steps needed to familiarize yourself with fBm.

Click the fBm module in the main area. It will now become highlighted, and the area at the right of the application, the inspector, will change.

In the inspector, you will see the name of the module, “fBm”, as well as several text boxes, labeled “pos”, “seed”, “octaves”, and so on. Note that these inputs correspond to the same inputs that can be seen in the graph view of the module. The “pos” textbox is grayed out, because it is connected to an output in the graph and receives its value from the “inputs” module. The other inputs are editable, because they are not connected to other modules.

Now, lets try to change some of the inputs of the fBm module. Scroll with your mouse wheel while hovering over the “gain” textbox. You should now be able to see the terrain change instantly. You should see something similar to Fig. A.5 happening. Set the gain back to 0.5 before you continue.

Now, try lowering the octaves value. As you do this, you should see “detail” disappearing from the terrain. What you are seeing, is the number of layers of noise



inspector: fBm

name: fBm
type: fbm2
fractional brownian motion
pos: 0 0
seed: 0
octav... 10
lacun... 2
gain: 0.500
comments:

Figure A.4: Click the fBm module so it can be edited in the inspector.

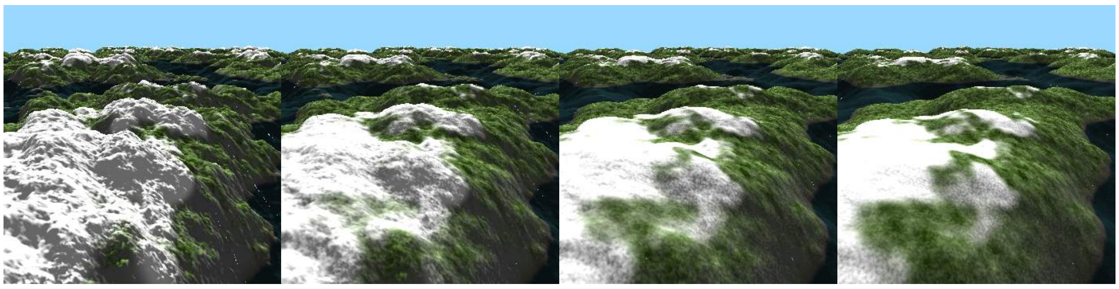


Figure A.5: Adjusting the gain of fBm

being modified. Each layer of noise has a higher frequency and lower amplitude than the one before, and that is why it is looking as if detail is being added and removed. The gain and lacunarity arguments describe the proportional change in amplitude and frequency changes between adjacent noise layers.

Be careful not to set the number of octaves too high, or the performance of you algorithm will suffer, and you may experience aliasing issues.

Now, you may recall that we wanted to create a coastal landscape. Currently, the landscape is equally rocky everywhere, and there is no water. To visualize how water would look, we will add a "max" module. On the right is a list of module types, click the plus sign next to the name "max". A new module should now appear in the graph. Click and drag the module to the right of the fBm module. Next, drag the green box labeled "result" from the fBm module to the "lhs" box of the "max" module. You have now connected the output of fBm to the input of max. Still, nothing will happen in the preview, since the output of max has still not been used for anything. Drag the output of max to the "height" input of the “outputs” module. Now you should notice the lower, blue areas of the terrain flattening out.

What happened now, was that at each coordinate we selected the highest value of our previous terrrain height, and “rhs”, which is zero. This results in all values previously below zero, to now be zero. By adjusting “rhs”, you can now move the “water” up and down.

Finally, for a more interesting terrain, try replacing the fBm module with a “hybridmultifractal”, or “ridgedmultifractal” module. The power of this tool comes from combining different terrains, though. So try out adding, scaling and clamping in various ways.

The mix module is also very useful, since it may be used to make gradual transitions between different terrains. This can be done by first designing two separate types of terrain, then place the mix module last in the graph, right before the output module. Let one terrain enter the x input, and another the y input. By adjusting the “a” input between 0 and 1, it is now possible to blend between the terrains. By inserting a low frequency, low octave fBm module, it is possible to slowly alternate between the terrain types.

A.5 User interface

Following are descriptions of the various features of the user interface and their functions.

A.5.1 Saving documents

Save your work by pressing the save icon, 💾, in the top left corner of the window.

A.5.2 Opening documents

There are two ways of opening documents:

- Click the open icon, 🗀, in the top left corner of the window. Note: this is currently not functioning due to a bug.
- Launch the program binary with the file to open as an argument. This is what happens when you double-click an nm.json-file. Alternatively, you may enter:

```
$ /path/to/nmgui /path/to/myterrain.nm.json
```

Or you may drag a document on top of the executable or the icon that starts the application.

A.5.3 Tabs

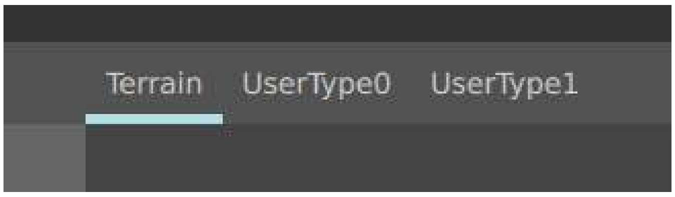


Figure A.6: Row of open tabs. The current tab, "Terrain", is highlighted.

Tabs can be opened to edit each of the user-defined module types. To open a new tab, click the pen icon, ✎, in the module type list next to the user type you want to edit.

A.5.4 Module type list

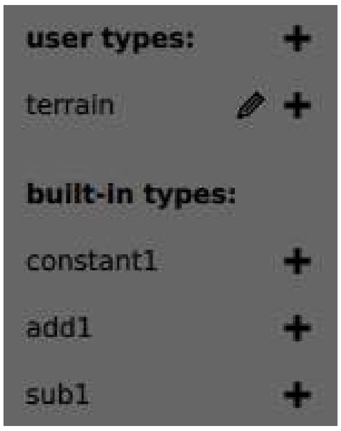


Figure A.7: Module type list

On the left hand side of the editor is the module type, showing the module types you may add to the current function graph. To add a module type, simply click the + icon next to the module type.

The module type list is divided into two parts. At the top is a list of all the user-defined module types. The one at the bottom contains built-in types. The user-defined types may be edited by clicking the edit icon, ✎. A new user type can by created by clicking the + next to the string "User types:".

If your screen resolution is too low for you to see all the module types, you may use the mouse scroll wheel to access the rest of the list.

A.5.5 Graph editor

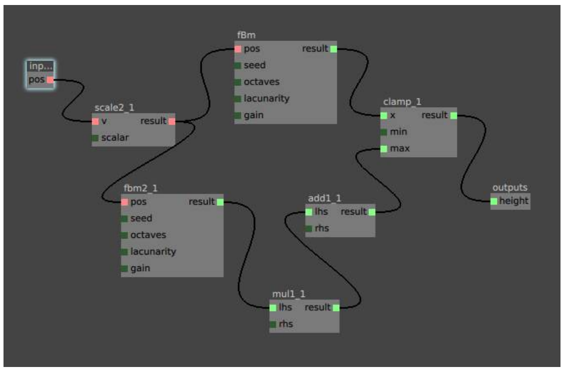


Figure A.8: The graph editor

The graph editor is the main area in the center of the window. This area lets you manipulate the structure of function graphs, as well as selecting which module to edit in the inspector (Appendix A.5.8).

Creating and destroying connections

Instances of module types (function types), i.e. modules, are shown as gray boxes that may be dragged around the main area. To connect an input to an output, simply drag from the output of one module to the input of another. Note that it is currently not possible to drag from an input to an output. To break a connection, simply click the input.

Color coding

Note that the inputs and outputs are color coded according to the dimensionality of the signal.

Green 1D signal (also known as scalars)

Red 2D signal (often used for heightmap positions)

Blue 3D signal

White 4D signal

Inputs and outputs have a darker shade of the color if they are disconnected.

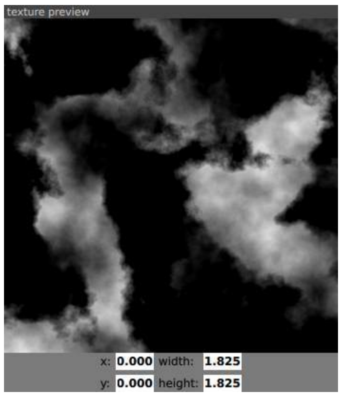
Panning

You may pan the graph if it gets to big to fit inside the window. Do this by clicking and dragging the graph background with the right or middle mouse button.

Selecting a module

To select a module, simply click it with the left mouse button. This will bring up the module inspector for editing the module.

A.5.6 2D preview



x: 0.000 width: 1.825
y: 0.000 height: 1.825

Figure A.9: 2D texture preview of the terrain.

The 2D texture preview shows a two-dimensional preview of the terrain where values of 0 and below are completely black, and values of 1 or above are completely white. Values between 0 and 1 have a varying shade of gray.

The currently previewed domain of the terrain is shown in the text boxes below the texture.

Double-click the preview to maximize it.

Translating the viewpoint

To preview another part of the terrain, click and drag the texture with the left mouse-button.

Scaling the preview

To preview a larger or smaller portion of the terrain, scroll with the mouse wheel above the preview. This will zoom in or out.

A.5.7 3D preview

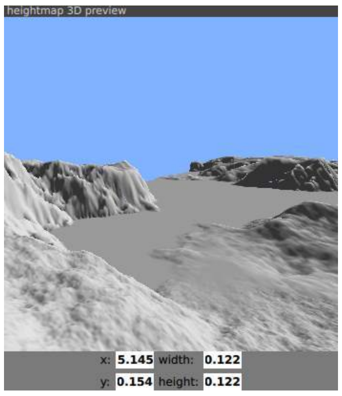


Figure A.10: 3D preview of the terrain.

The 3D preview shows a 3D rendering of the terrain. The camera angle and position may be controlled using the keyboard. The following keybindings are defined:

W Move forward

S Move backward

A Move camera left

D Move camera right

Left arrow Rotate camera counter-clockwise

Right arrow Rotate camera clockwise

Up arrow Rotate camera up

Down arrow Rotate camera down

Click-and-drag with left mouse button Rotate camera freely

Inside the preview, you may click-and-drag to control the camera in a way that is common in many first-person games. Dragging from left to right changes the horizontal direction of the camera. Dragging up or down lifts or lowers the camera.

Maximizing the preview

Double-click the preview to maximize it.

Controlling texturing

The preview comes with an option for turning on and off texturing for the terrain. There are currently two modes:

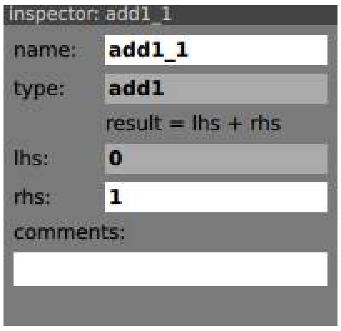
Texturing on An automatic procedural texturing of the terrain based on terrain height. Snow appears on values close to or above height=1, and a blue water color is applied where height<0. Normals are also perturbed slightly to make surfaces less uniform. This feature is not very well optimized, especially in full screen. Consider turning it off, if you are experiencing low frame rates.

Texturing off Texturing is a simple diffuse white. This is not a very interesting preview, but it makes it easier to see the shape of the terrain, and it is also very fast.

There is a checkbox at the bottom of the 3D preview that lets you toggle texturing.

A.5.8 Inspector

The inspector lets you show information about and edit selected entities. Currently, only modules (instances of a module type) are editable in the inspector. Editing *module type* attributes, such as name and description, will be added later.



Inspector: add1_1
name: add1_1
type: add1
result = lhs + rhs
lhs: 0
rhs: 1
comments:

Figure A.11: The inspector showing an add1 module, a one-dimensional addition module. Only the "rhs" (right-hand-side) value is editable. "lhs" is gray and cannot be edited because it is currently connected to the output of another module.

Changing name and comments

The name of the module may be edited by editing the text in the name field. This will change how the module appears in the graph editor.

The comments field is a place where you may write whatever you want. Use it if you feel that something about the graph or the parameter values may need to be explained.

Tuning parameter values

If an input is not connected to an output, it may be assigned a constant value.

Parameter values may be entered by using the keyboard, but this is not the recommended way of tuning parameters.

Scrolling the mouse-wheel on a component adjusts its value. Scrolling upwards increases the value by 10 percent, scrolling down decreases the value by 10 percent. When you adjust a parameter in this way, you will clearly notice its effects on the terrain by keeping an eye at the real-time preview.

A.6 Module types

Table A.1 contains a list of the available module types, as well as a mathematical definition for their behavior. Some module types were not added to the table because their definitions are too long to fit inside the table cells.

“fbm2”, “ridgedmultifractal”, and “hybridmultifractal” are three different variants of scaled and added noise. The definitions for these functions may be found in the project source code, and are a relatively direct port of the original source code of Musgrave [3].

A.7 Using the library to generate terrains online

This section covers how the library can be used to facilitate generation of terrain data during run-time.

A.7.1 Loading a graph from JSON

This section assumes that you have already created a terrain function using the Noise Modeler application and stored it in an “nm.json” file.

First, load the “nm.json” file into a string using a file input/output library for your operating system.

```
std::string serializedGraph = readFileToString("terrain.nm.json"↩
    );
```

Then parse the json into a TypeManager.

```
nm::Parser parser;
nm::optional<std::unique_ptr<nm::Parser>> maybeTypeManager = ↩
    parser.parse(serializedGraph);

//check if parsing succeeded
if(!maybeTypeManager){
    //TODO handle errors
    exit(EXIT_FAILURE);
}

//create an alias to the typeManager for more convenient access
nm::TypeManager &typeManager = *(*maybeTypeManager);
```

After this, you may use the `typeManager` alias to access any "user types" defined in the file.

A.7.2 Generating a GLSL elevation function from a user type

Assuming you have already parsed a document into a `TypeManager`, you may use the following code to generate a GLSL function:

```
//we will be using the nm namespace frequently
using namespace nm;

//get the relevant graph
ModuleType* terrainModuleType = typeManager.getUserType("terrain↩
    ");

//get the graph of the module
Graph* graph = terrainModuleType->getGraph();

//get the input and output for your function
InputLink* input = graph->getModule("inputs")->getInput("pos");
OutputLink* output = graph->getModule("outputs")->getOutput("↩
    height");

//generate a function called "elevation".
std::string glslSourceCode = glsl::GlslGenerator::↩
    compileToGlslFunction(*input, *output, "elevation");
```

This will create a string, `glslSourceCode`, containing the GLSL source code for a function called `elevation`, as well as the functions it depends on, such as noise.

Note that the code above does not handle cases where `pos` and `height` are not valid inputs and outputs of the node. Always check for null pointers before dereferencing.

When you create your GLSL shader, concatenate the generated source code with your own shader code. You may then call the `elevation` function in your own code like this:

```
vec2 pos = vertexPosition.xy;
float height;
elevation(pos, height);
gl_Position = vec4(vertexPosition.xy, height, 1);
```

A.7.3 Dynamically creating heightmap textures using a GLSL function

Once you have created a GLSL function, you can either use it directly when rendering, i.e. by offsetting heights in the vertex shader, or you may use the function to generate a heightmap which you can pass on to your game engine.

You may do the following steps:

1. Draw two triangles into a frame buffer object of the appropriate resolution, filling it completely.
2. In the fragment shader, sample the generated elevation function at the desired position.
3. Transfer the frame buffer object back to the CPU, using the OpenGL function `glReadPixels`.
4. Change the pixel format (optional).
5. Pass the pixel data over to your game engine.

A similar process has been followed to create the texture preview in the Noise Modeler application. Its source code is publicly available and may be regarded as an example of how to use the API.

There is also a minimal example of how to generate terrain heightmaps in the benchmark application for the library, which uses a window-less OpenGL context to generate height data, and transfers it back to CPU memory.

Name	Inputs	Outputs	Definition
constant<D>	$value(D)$	$value(D)$	$value = value$
add<D>	$lhs(D), rhs(D)$	$result(D)$	$result = lsh + rhs$
sub<D>	$lhs(D), rhs(D)$	$result(D)$	$result = lsh - rhs$
mul<D>	$lhs(D), rhs(D)$	$result(D)$	$result = lsh \cdot rhs$
scale<D>	$v(D), scalar$	$result(D)$	$result = scalar \cdot v$
mod	$dividend, divisor$	$result$	$result \equiv dividend \mod divisor$
min	a, b	$result$	$result = \left\{ \begin{array}{ll} a & \text{if } a < b \\ b & \text{if } a \geq b \end{array} \right\}$
max	a, b	$result$	$result = \left\{ \begin{array}{ll} a & \text{if } a \geq b \\ b & \text{if } a < b \end{array} \right\}$
abs	$source$	$result$	$result = \|source\|$
clamp	x, a, b	$result$	$result = \left\{ \begin{array}{ll} x & \text{if } a < x < b \\ a & \text{if } x \leq a \\ b & \text{if } x \geq b \end{array} \right\}$
step	$value, edge$	$result$	$result = \left\{ \begin{array}{ll} 0 & \text{if } value < edge \\ 1 & \text{if } value \geq edge \end{array} \right\}$
smoothstep	$value, minedge, maxedge$	$result$	$value^* = clamp(\frac{value - minedge}{maxedge - minedge}, 0, 1)$ $result = value^{*2}(3 - 2 \cdot value^*)$
mix	x, a, b	$result$	$result = smootstep(x, 0, 1) \cdot a$ $+ smoothstep(x, 1, 0) \cdot b$
demux2	$m(2)$	x, y	$x = m.x$ $y = m.y$
demux3	$m(3)$	x, y, z	$x = m.x$ $y = m.y$ $z = m.z$
demux4	$m(4)$	x, y, z, w	$x = m.x$ $y = m.y$ $z = m.z$ $w = m.w$
mux2	x, y	$m(2)$	$m.x = x$ $m.y = y$
mux3	x, y, z	$m(3)$	$m.x = x$ $m.y = y$ $m.z = z$
mux4	x, y, z, w	$m(4)$	$m.x = x$ $m.y = y$ $m.z = z$ $m.w = w$
fbm2	$pos(2), seed, octaves,$ $lacunarity, gain$	$result$	Too long for table.
hybridmultifractal	$pos(2), seed, octaves,$ $lacunarity, h, offset$	$result$	Too long for table.
ridgedmultifractal	$pos(2), seed, octaves,$ $lacunarity, h, offset, gain$	$result$	Too long for table.

Table A.1: Built-in module types. Function<D> denotes that there is one definition for each dimensionality. "Variable(D)" denotes a D-dimensional vector, if there is no parenthesis, the type is a scalar.

Appendix B

Poster Submission for SIGGRAPH 2014

NTNU

Framework for Real-Time Editing of Endless Procedural Terrains

Johan K. Helsing and Anne C. Elster

HPC-Lab
Computer & Info. Science
Norwegian University of Science and Technology
http://research.idi.ntnu.no/hpc-lab

Abstract

- Design endless procedural terrain in real time while continuously viewing the changes. The designed terrain can be used by game engines without losing the benefits of procedural generation. Terrain can be generated online, on the GPU, while your game is running, rather than be exported as a heightmap.

Motivation

- Procedural content generation is an important feature of many recent games. Many games feature endless virtual worlds, or worlds that are different each time a game new game is started.
- Existing procedural terrain editing tools loose most of the benefits of procedural generation by requiring the terrain to be exported to a non-procedural format before being used by a game engine. This makes the tools unusable by endless world games, such as Minecraft.
- Noise synthesis libraries such as libnoise and the Accidental Noise Library have code-only interfaces, making rapid prototyping and experimentation difficult.
- A reusable middleware for terrain generation can support non-essential, but desired features, such as evaluation on the GPU. And enable reuse of terrain designs across projects.

Our Method

- Terrain represented as heightmap function, $f(x,y)$, which returns height at coordinate (x,y)
- Through functional composition, noise and other algorithms are combined to create a heightmap function. I.e. inputs of one function can be bound to the outputs of one or multiple other functions.
- Functional composition represented as DAG (directed acyclic graph).
- JSON used to serialize graphs.
- Noise Modeler, our grahical tool, makes it possible to intuitively edit function graphs.
- nmlib, our library, parses serialized function graphs, and can be used to query for elevation data while the game is running.
- All computations are performed by the GPU.

Flow-graph Editor

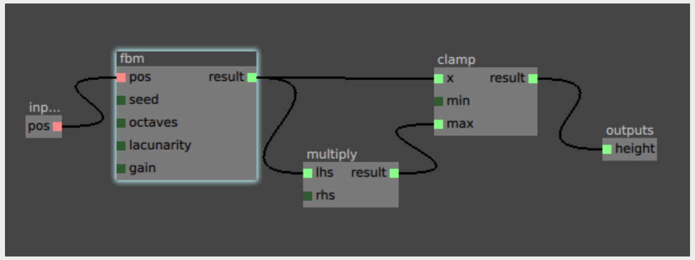


Figure: Build a function graph by dragging and dropping inputs and outputs of well-known noise and math functions.

Real-time Preview

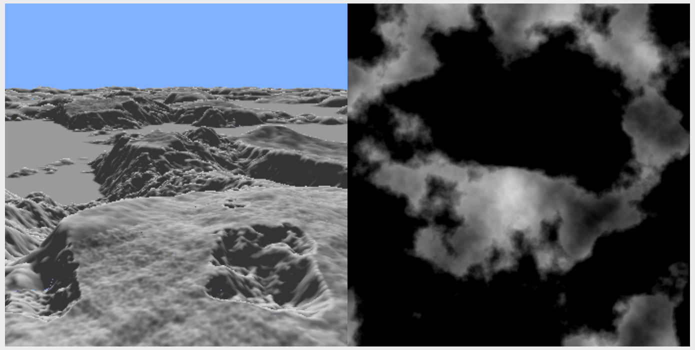


Figure: Navigate and explore the resulting terrain in a real-time 3D preview, or as a texture that can be panned and zoomed

Design Your Own High-level Modules

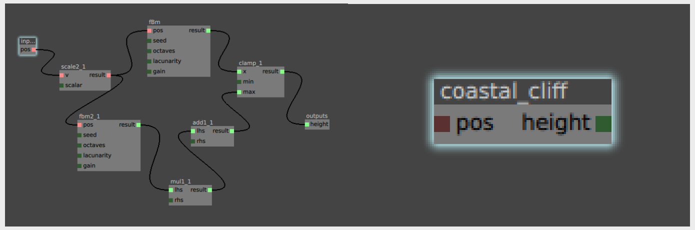


Figure: Encapsulate complicated sub-graphs as new module types

Combine Different Biomes for a Richer Landscape

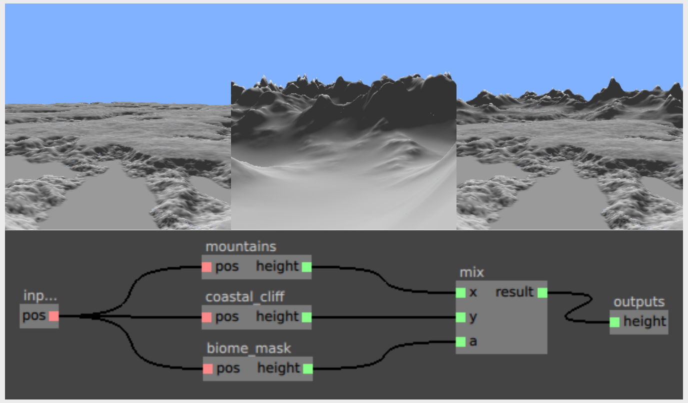


Figure: Different biomes can be combined using a high-level mask, such as fractional-brownian motion with a low frequency and few octaves.

Results

- Working GPU-based prototype is available. Terrain can be generated and explored in real-time even without caching elevation data.
- Function graphs serialized as JSON consume only a few kilobytes of storage.
- OpenGL 3.0+ support is the only requirement for the evaluation platform.

Future Work

- Improve preview rendering with diffuse, specular, and ambient textures, as well as optimizations such as caching, frustum culling, and continuous level-of-detail.
- Preview support for other terrain types, e.g. voxel terrain and vector displacement terrain.
- Evaluation on additional platforms.
- Plugins for common game engines, such as Unity and Unreal.

References

[1] David S Ebert, F Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steve Worley.
Texturing & modeling: a procedural approach.
Morgan Kaufmann, 2003.

[2] Ruben M Smelik, Tim Tutenel, Rafael Bidarra, and Bedrich Benes.
A survey on procedural modelling for virtual worlds.
In *Computer Graphics Forum*. Wiley Online Library, 2014.

Appendix C

Benchmark code

Listing C.1: libnoise, uncached benchmark

```cpp
#include <noise/noise.h>

#include <iostream>
#include <vector>

using namespace std;
using namespace noise;

int main(int argc, char *argv[]){
  module::Perlin perlin;
  perlin.SetOctaveCount(8.0);

  //use an interpolation function with continuous first order ↩
     derivative
  perlin.SetNoiseQuality(QUALITY_STD);

  module::Const half;
  half.SetConstValue(0.5);

  module::Const quarter;
  quarter.SetConstValue(0.25);

  module::Const threeQuarter;
  threeQuarter.SetConstValue(0.75);

  module::Multiply multiply;
  multiply.SetSourceModule(0, perlin);
  multiply.SetSourceModule(1, quarter);

  module::Add add;
  add.SetSourceModule(0, threeQuarter);
  add.SetSourceModule(1, multiply);

  //clamp module does not support non-const limits,
  //so we have to use a max and a min module
  module::Max max;
  max.SetSourceModule(0, perlin);
  max.SetSourceModule(1, quarter);

  module::Min min;
  min.SetSourceModule(0, max);
  min.SetSourceModule(1, multiply);

  //create buffers to store each row
  vector<vector<float> > buffers(8192);
  for(int i=0; i<8192; ++i){
    buffers[i].reserve(8192);
  }

  for(int i=0; i<8192; ++i){
    for(int j=0; j<8192; ++j){
      float x=i*0.01,y=j*0.01,z=0;
      buffers[i][j]=min.GetValue(x,y,z);
    }
  }

  return EXIT_SUCCESS;
}
```

Listing C.2: libnoise, cached benchmark

```
#include <noise/noise.h>

#include <iostream>
#include <vector>

using namespace std;
using namespace noise;

int main(int argc, char *argv[]){
  module::Perlin perlin;
  perlin.SetOctaveCount(8.0);

  //use an interpolation function with continuous first order ↩
     derivative
  perlin.SetNoiseQuality(QUALITY_STD);

  //cache the perlin module to avoid computing the same result ↩
```

```
     twice in a row
  module::Cache perlinCached;
  perlinCached.SetSourceModule(0, perlin);

  module::Const half;
  half.SetConstValue(0.5);

  module::Const quarter;
  quarter.SetConstValue(0.25);

  module::Const threeQuarter;
  threeQuarter.SetConstValue(0.75);

  module::Multiply multiply;
  multiply.SetSourceModule(0, perlinCached);
  multiply.SetSourceModule(1, quarter);

  module::Add add;
  add.SetSourceModule(0, threeQuarter);
  add.SetSourceModule(1, multiply);

  //clamp module does not support non-const limits,
  //so we have to use a max and a min module
  module::Max max;
  max.SetSourceModule(0, perlinCached);
  max.SetSourceModule(1, quarter);

  module::Min min;
  min.SetSourceModule(0, max);
  min.SetSourceModule(1, multiply);

  //create buffers to store each row
  vector<vector<float> > buffers(8192);
  for(int i=0; i<8192; ++i){
    buffers[i].reserve(8192);
  }

  for(int i=0; i<8192; ++i){
    for(int j=0; j<8192; ++j){
      float x=i*0.01,y=j*0.01,z=0;
      buffers[i][j]=min.GetValue(x,y,z);
    }
  }

  return EXIT_SUCCESS;
}
```

Listing C.3: ANL, uncached benchmark

```
fbm=anl.CImplicitFractal(anl.FBM, anl.SIMPLEX, anl.CUBIC, 8, 2, true)
multiply=anl.CImplicitMath(anl.MULTIPLY, fbm, 0.25)
add=anl.CImplicitMath(anl.ADD, multiply, 0.75)
clamp=anl.CImplicitClamp(fbm, 0.25, add)

ad=anl.CImplicitBufferImplicitAdapter(clamp, anl.SEAMLESS_NONE, anl.SMappingRanges(0,8,0,8,0,8), false, 0)

i=anl.CArray2Dd()
i:resize(8192,8192)

ad:get(i)
```

Listing C.4: ANL, cached benchmark

```
fbm=anl.CImplicitFractal(anl.FBM, anl.SIMPLEX, anl.CUBIC, 8, 2, true)
fbmCached=anl.CImplicitCache(fbm)
multiply=anl.CImplicitMath(anl.MULTIPLY, fbmCached, 0.25)
add=anl.CImplicitMath(anl.ADD, multiply, 0.75)
clamp=anl.CImplicitClamp(fbmCached, 0.25, add)

ad=anl.CImplicitBufferImplicitAdapter(clamp, anl.SEAMLESS_NONE, anl.SMappingRanges(0,8,0,8,0,8), false, 0)

i=anl.CArray2Dd()
i:resize(8192,8192)

ad:get(i)
```

Listing C.5: nmlib benchmark

```
#include <GL/glew.h>
#include <GLFW/glfw3.h>

#include <cstdlib>
#include <iostream>
#include <sstream>
#include <fstream>
#include <array>
#include <ctime>
#include <chrono>

#include <nmlib/serialization.hpp>
```

```
#include <nmlib/model.hpp>
#include <nmlib/codegeneration/glsl/glslgenerator.hpp>

using namespace std;
string createElevationFunction(){
  //get json from file
  string filename = "terrain.nm.json";
  ifstream t(filename);
  if(!t || !t.good()){
    cerr << "Couldn't open terrain.nm.json\n";
    glfwTerminate();
    exit(EXIT_FAILURE);
  }
  stringstream buffer;
  buffer << t.rdbuf();
  string json = buffer.str();

  //parse json
  nm::Parser parser;
  auto maybeTypeManager = parser.parseDocument(json);
  if(!maybeTypeManager){
    cerr << "Error parsing terrain document\n";
    glfwTerminate();
    exit(EXIT_FAILURE);
  }

  nm::TypeManager& typeManager = **maybeTypeManager;
  auto userType = typeManager.getUserType(0);
  auto inputs = userType->getGraph()->getModule("inputs");
  auto outputs = userType->getGraph()->getModule("outputs");
  auto posInput = inputs->getInput(0);
  auto heightOutput = outputs->getOutput(0);

  string elevationFunction = nm::glsl::GlslGenerator::↩
      compileToGlslFunction(
       *posInput, *heightOutput, "elevation");

  return elevationFunction;
}

void errorCallback(int error, const char* description){
  cerr << "GLFW error: " << description << endl;
}

int main(int argc, char *argv[]){
  typedef std::chrono::duration<int,std::milli> ms_t;
  using chrono::system_clock;
  auto start = system_clock::now();
  //width of the texture
  const int resolution = 8192;
  //number of chunks to store the texture in
  //this may be useful on machines were it is not possible
  //to get 8192*8192*4*sizeof(float) contiguous memory.
  //each chunk corresponds to a glReadPixels call
  const int chunks = 1;

  glfwSetErrorCallback(errorCallback);

  if(!glfwInit())exit(EXIT_FAILURE);

  //create an OpenGL context
  //make the window stay hidden, since we do not need it
  glfwWindowHint(GLFW_VISIBLE, GL_FALSE);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  GLFWwindow* window = glfwCreateWindow(640, 480, "nmlib benchmark", NULL, NULL);
  if(!window){
    glfwTerminate();
    exit(EXIT_FAILURE);
  }
  glfwMakeContextCurrent(window);

  if(glewInit() != GLEW_OK){
    cerr << "Error initializing GLEW\n";
    glfwTerminate();
    exit(EXIT_FAILURE);
  }

  cout << "Version string for OpenGL context: " << glGetString(GL_VERSION) << endl;
  cout << "GLSL version: " << glGetString(GL_SHADING_LANGUAGE_VERSION) << endl;

  if(!GLEW_EXT_framebuffer_object){
    cout << "Error: no extension GL_EXT_framebuffer_object." << endl;
    glfwTerminate();
    exit(EXIT_FAILURE);
  }

  if(!GLEW_ARB_color_buffer_float){
    cerr << "Error: no extension ARB_color_buffer_float." << endl;
    glfwTerminate();
    exit(EXIT_FAILURE);
  }

  //create fbo (off-creen framebuffer)
  GLuint fb;
  glGenFramebuffers(1, &fb);

  glBindFramebuffer(GL_FRAMEBUFFER, fb);

  int maxtexsize;
  glGetIntegerv(GL_MAX_TEXTURE_SIZE, &maxtexsize);
  cout << "Max texture size: " << maxtexsize << endl;
  if(maxtexsize<resolution){
    cerr << "Max texture size too small for resolution: " << resolution << endl;
  }

  glClampColor(GL_CLAMP_READ_COLOR, GL_FALSE);
  glClampColor(GL_CLAMP_VERTEX_COLOR, GL_FALSE);
  glClampColor(GL_CLAMP_FRAGMENT_COLOR, GL_FALSE);

  GLuint color_tex;
  glGenTextures(1, &color_tex);
  glBindTexture(GL_TEXTURE_RECTANGLE, color_tex);
  glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_RECTANGLE, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  //NULL means reserve texture memory, but texels are undefined
  glTexImage2D(GL_TEXTURE_RECTANGLE, 0, GL_RGBA, resolution, resolution, 0, GL_RGBA, GL_UNSIGNED_BYTE, NULL);

  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_RECTANGLE, color_tex, 0);

  if(glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE){
    cerr << "Framebuffer incomplete\n";
    glfwTerminate();
    exit(EXIT_FAILURE);
  }

  GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
  {
    if(0==vertexShader){
      cerr << "Error creating vertex shader";
      glfwTerminate();
      exit(EXIT_FAILURE);
    }
    const GLchar* shaderCode = ""
      "#version 130\n"
      "uniform vec4 domain; //{x, y, width/2, height/2}\n"
      "in vec2 vertices;\n"
      "out vec2 coords;\n"
      "void main() {\n"
      "    gl_Position = vec4(vertices.x,vertices.y,0,1);\n"
      "    coords = vec2(vertices.x, -vertices.y)*domain.zw + domain.xy;\n"
      "}\n"
      ;
    const GLchar * codeArray[] = {shaderCode};
    glShaderSource(vertexShader, 1, codeArray, NULL);
    glCompileShader(vertexShader);
    GLint result;
    glGetShaderiv(vertexShader, GL_COMPILE_STATUS, &result);
    if(GL_FALSE==result){
      cerr << "Vertex shader compilation failed!\n";
      GLint logLen;
      glGetShaderiv(vertexShader, GL_INFO_LOG_LENGTH, &logLen);
      if(logLen>0){
        char *log = new char[logLen];
        GLsizei written;
        glGetShaderInfoLog(vertexShader, logLen, &written, log);
        cerr << "Shader log:\n" << log << endl;
        delete log;
      }
    }
}

  GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
  {
    if(0==fragmentShader){
      cerr << "Error creating vertex shader";
      glfwTerminate();
      exit(EXIT_FAILURE);
    }
    const GLchar* versionString = "#version 130\n";
    string elevationFunction = createElevationFunction();
    const GLchar* mainFunction = ""
      "in vec2 coords;\n"
      "void main() {\n"
      "    float height;\n"
      "    elevation(coords, height);\n"
      "    gl_FragColor = vec4(height, height, height, 1);\n"
      "}\n"
      ;
    const GLchar * codeArray[] = {versionString, elevationFunction.c_str(), mainFunction};
```

```cpp
    glShaderSource(fragmentShader, 3, codeArray, NULL);
    glCompileShader(fragmentShader);
    GLint result;
    glGetShaderiv(fragmentShader, GL_COMPILE_STATUS, &result);
    if(GL_FALSE==result){
      cerr << "Fragment shader compilation failed!\n";
      GLint logLen;
      glGetShaderiv(fragmentShader, GL_INFO_LOG_LENGTH, &logLen);
      if(logLen>0){
        char *log = new char[logLen];
        GLsizei written;
        glGetShaderInfoLog(fragmentShader, logLen, &written, log);
        cerr << "Shader log:\n" << log << endl;
        delete log;
      }
    }
  }

  //link shader program
  GLuint programHandle = glCreateProgram();
  glAttachShader(programHandle, vertexShader);
  glAttachShader(programHandle, fragmentShader);
  glLinkProgram(programHandle);

  GLint linkStatus;
  glGetProgramiv(programHandle, GL_LINK_STATUS, &linkStatus);
  if(GL_FALSE == linkStatus){
    cerr << "Failed to link shader program\n";
    glfwTerminate();
    exit(EXIT_FAILURE);
  }
  glUseProgram(programHandle);

  //setup vbo
  array<float, 8> vertices = {
    -1,-1,
     1,-1,
    -1, 1,
     1, 1
  };
  GLuint vboId;
  glGenBuffers(1, &vboId);
  glBindBuffer(GL_ARRAY_BUFFER, vboId);
  glBufferData(GL_ARRAY_BUFFER, 4*2*sizeof(float), &vertices[0], GL_STATIC_DRAW);
  glVertexAttribPointer((GLuint)0, 2, GL_FLOAT, GL_FALSE, 0, 0);
  glEnableVertexAttribArray(0);
```

```

//set uniforms
array<float, 4> domain = {0,0,4,4};
GLint domainLoc = glGetUniformLocation(programHandle, "domain");
glUniform4fv(domainLoc, 1, &domain[0]);

glViewport(0,0,resolution,resolution);

auto beforeDrawArrays = system_clock::now();
//start drawing call
glClearColor(0, 0, 0, 1);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4); //four vertices to draw the two triangles
auto afterDrawArrays = system_clock::now();

//release program
glDeleteProgram(programHandle);

//create buffers to store chunks subtextures
vector<vector<float> > buffers(chunks);
for(int i=0; i<chunks; ++i){
  buffers[i].reserve(resolution*resolution/chunks*4);
}
for(int i=0; i<chunks; ++i){
  cout << "Loading chunk " << i << "...";
  auto beforeReadPixels = system_clock::now();
  glReadPixels(0, i*(resolution/chunks), resolution, resolution/chunks, GL_RED, GL_FLOAT, &buffers[i][0]);
  auto afterReadPixels = system_clock::now();
  ms_t ms = chrono::duration_cast<ms_t>(afterReadPixels-beforeReadPixels);
  cout << " " << ms.count() << " ms" << endl;
}
//print a random value, to trick the compiler into not optimizing out
//parts of the buffers
cout << "random value in pixel array: " << buffers[rand()%chunks][rand()%(resolution*resolution/chunks)] << endl;

auto finish = system_clock::now();
{
  ms_t milliseconds = chrono::duration_cast<ms_t>(finish-start);
  cout << "Finished after " << milliseconds.count() << " ms" << endl;
}

```

```
  {
    ms_t milliseconds = chrono::duration_cast<ms_t>(finish-↩
       beforeDrawArrays);
    cout << "Without context creation and compilation: " << ↩
       milliseconds.count() << " ms" << endl;
  }

  glfwDestroyWindow(window);
  glfwTerminate();
}
```

Appendix D

Noise Modeler Library API Reference

Noise Modeler Library

0.1

Generated by Doxygen 1.8.7

Sun Jun 29 2014 12:37:16

Contents

1 Noise Modeler Library Documentation

Introduction

Welcome to the reference for the noise modeler library, nmlib.

The Noise Modeler Library is a library for specifying content generating procedural functions based on noise in a platform independent manner.

The prime example of such content, is terrain heightmap generation. With the help of this library, it is possible represent terrains that are several times the size of the earth using only a few kilobytes. This representation may be expanded during run-time in a game at high speed by utilizing the GLSL code generation feature.

Other content may for example be vegetation density maps, air humidity, and area distributions among factions in a game.

Models for nmlib may be created using the Noise Modeler GUI application, which is a real-time editor for endless procedural terrains. This application is documented by its own user guide.

Documentation overview

The library is divided into four modules, and so is the documentation:

- model is the core part of the library, everything depends on this.
- serialization is concerned with serialization and parsing of models.

- codegeneration is responsible for generating code, such as GLSL to evaluate functions.
- util contains utility functions for the library.

2 Module Index

2.1 Modules

Here is a list of all modules:

3 Hierarchical Index

3.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

4 Class Index

4.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

5 File Index

5.1 File List

Here is a list of all documented files with brief descriptions:

6 Module Documentation

6.1 codegeneration

Code generation module.

Classes

- class nm::BodyGenerator

  Generates a module body.
- class nm::CompositeModuleGenerator

  Generator for modules having composite module types.
- class nm::ConcreteModuleGenerator

  The ConcreteModuleGenerator class.
- class nm::DefaultsGenerator

  Generates the definitions and default values module inputs.
- class nm::FunctionCallBodyGenerator

  The FunctionCallBodyGenerator class.
- class nm::glsl::GlslGenerator

Generates glsl code to evaluate function graphs using OpenGL 3.0 shaders.

- class nm::IdGenerator

  The IdGenerator class.
- class nm::InlineGenerator

  Abstract base class for code generators.
- class nm::ModuleGenerator

  The ModuleGenerator class.
- class nm::SimpleBodyGenerator

  The SimpleBodyGenerator class.

6.1.1 Detailed Description

Code generation module.

InlineGenerator is the most important class. Subclass it to implement code generation for a new language.

glsl::GlslGenerator implements InlineGenerator for GLSL

6.2 model

Graph representation of a procedural generator.

Files

- file model.hpp

    The model module.

Classes

- class nm::Graph

    A graph of Modules.
- class nm::InputLink

    Describes which output of which node a ModuleInput of a specific Module is connected to.
- class nm::Module

    An instantiated ModuleType. A node in a function graph.
- class nm::ModuleInput

    Describes one of a ModuleType's inputs (name, SignalType, default value).
- class nm::ModuleOutput

    Describes one of a ModuleType's outputs (name, SignalType)
- class nm::ModuleType

    Describes a recipe for a module and its inputs and outputs.
- class nm::OutputLink

    an output of a Module
- class nm::SignalType

    Describes a the dimensionality of a signal Can be extended to include other type information such as distinctions between doubles, flots and ints as well.
- class nm::SignalValue

    A vector of floats that can be set as the unlinked value of InputLinks.
- class nm::TypeManager

    The top-level entity of a noise model. Encapsulates several user types as well as built-in types.

6.2.1 Detailed Description

Graph representation of a procedural generator.

The model contains classes for representing module types and graphs of modules.

The top-level entity, is the TypeManager, which may contain several ModuleTypes. A ModuleType represents a mathematical function or algorithm. It is either a primitive type defined by the library, or a composite type described as a Graph of Modules.

When a ModuleType is described by a Graph of Modules, it is called a composite type.

A Graph is a directed acyclic graph of function calls (Modules). It is a similar concept to what is commonly known as an expression tree.

Each node in the Graph, is called a Module. A module represents a function call, and has a corresponding Module↩Type, and information about the inputs and outputs of the function call. A Module has several InputLinks that may be connected to OutputLinks of other Modules.

Here is an example of how the expression `abs(0.25+fbm(x,y))` may be created using the model:

```
//A type manager holds information about module types
TypeManager typeManager;

//Initialize common module types, such as add, abs, and fbm2d
typeManager.initBuiltinTypes();

//create a module type representing our terrain function
ModuleType *terrainType = typeManager.createModuleType("terrain")

//Add inputs and outputs to the module type
//create a 2D input, "pos"
terrainType->addInput("pos", SignalType{2})
//create a 1D output, "height"
terrainType->addOutput("position", SignalType{2})

//get a handle for the graph of the module type
Graph *graph = terrainType.getGraph();
Module *inputs = graph->getModule("inputs");
Module *outputs = graph->getModule("outputs");

//create modules for the fbm (2d), add (1d), abs (1d)
Module* fbmModule = graph->createModule(*typeManager.getBuiltinType("fbm2"));
Module* addModule = graph->createModule(*typeManager.getBuiltinType("add1"));
Module* absModule = graph->createModule(*typeManager.getBuiltinType("abs"));

//connect fbm to position
fbmModule->getInput("pos")->link(*inputs->getOutput("pos"));

//connect add input to fbm output
addModule->getInput("lhs")->link(*fbmModule->getOutput("result"));
//set the right hand side to 0.25
addModule->getInput("rhs")->setUnlinkedValue(SignalValue({0.25}));

//set abs source to add output
absModule->getInput("source")->link(*addModule->getOutput("result"));

//connect abs output to module type output
outputs->getInput("height")->link(*absModule->getOutput("result"));

//we now have a complete model for a heightmap generating
//module type: terrainType
//use the terrainType to evaluate some terrain, or serialize it to disk
```

6.3 serialization

Everything related to converting graphs to/from strings.

Files

- file serialization.hpp

  The serialization module.

Classes

- class nm::Parser

  Converts json strings to TypeManagers.
- class nm::Serializer

  Serializes a TypeManager to a JSON string.

6.3.1 Detailed Description

Everything related to converting graphs to/from strings.

Each of the classes, Parser and Serializer are documented separately.

6.4 util

utility functions and classes needed by other nmlib modules.

Files

- file util.hpp

  Utility headers.

Classes

- class nm::NonCopyable

  A super-class for non-copyable classes.
- class nm::UserDataProvider

  Base class for stuff that needs to provide user data in form of a void$*$ pointer.

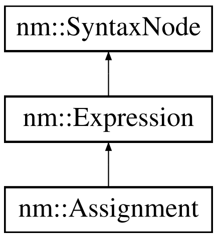
6.4.1 Detailed Description

utility functions and classes needed by other nmlib modules.

7 Class Documentation

7.1 nm::Assignment Struct Reference

Inheritance diagram for nm::Assignment:



Public Member Functions

- **Assignment** (std::unique_ptr< Variable > id, std::unique_ptr< Expression > value)
- **Assignment** (std::string l, std::string r)
- virtual void **gen** (InlineGenerator &g, std::ostream &out) override

Public Attributes

- std::unique_ptr< Variable > **lhs**
- std::unique_ptr< Expression > **rhs**

The documentation for this struct was generated from the following file:

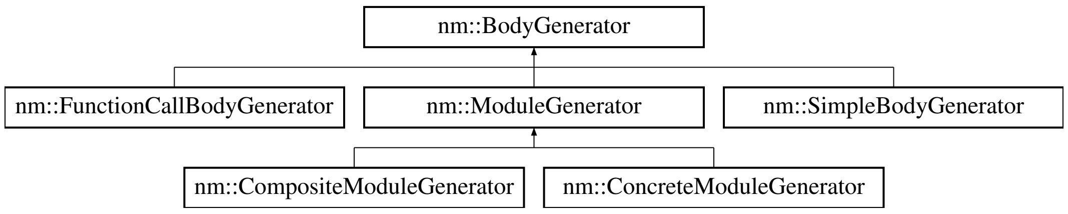
- codegeneration/inlinegenerator.hpp

7.2 nm::BodyGenerator Class Reference

Generates a module body.

`#include <bodygenerator.hpp>`

Inheritance diagram for nm::BodyGenerator:


nm::BodyGenerator
nm::FunctionCallBodyGenerator
nm::ModuleGenerator
nm::SimpleBodyGenerator
nm::CompositeModuleGenerator
nm::ConcreteModuleGenerator

Public Member Functions

- virtual void **generateBody** (InlineGenerator &gen, std::ostream &out)=0

7.2.1 Detailed Description

Generates a module body.

The documentation for this class was generated from the following file:

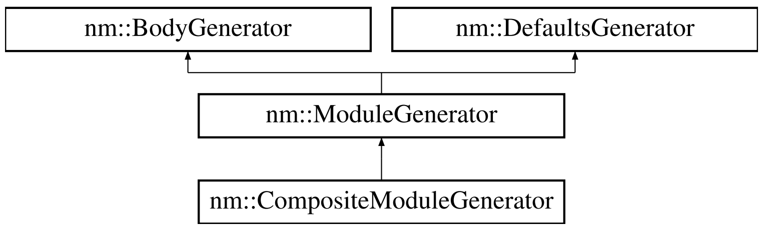
- codegeneration/bodygenerator.hpp

7.3 nm::CompositeModuleGenerator Class Reference

Generator for modules having composite module types.

`#include <compositemodulegenerator.hpp>`

Inheritance diagram for nm::CompositeModuleGenerator:



Public Member Functions

- **CompositeModuleGenerator** (const Module &module)
- void **generateDefaults** (InlineGenerator &gen, std::ostream &out) override
- void **generateBody** (InlineGenerator &gen, std::ostream &out) override

7.3.1 Detailed Description

Generator for modules having composite module types.

The documentation for this class was generated from the following files:

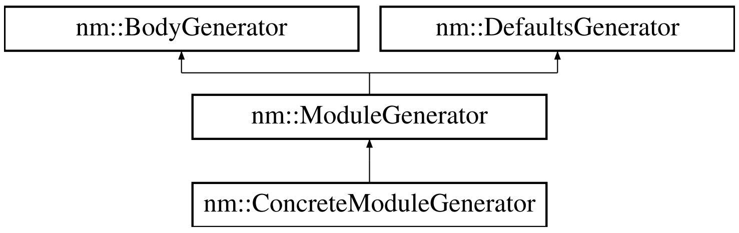
- codegeneration/compositemodulegenerator.hpp
- codegeneration/compositemodulegenerator.cpp

7.4 nm::ConcreteModuleGenerator Class Reference

The ConcreteModuleGenerator class.

`#include <concretemodulegenerator.hpp>`

Inheritance diagram for nm::ConcreteModuleGenerator:



Public Member Functions

- **ConcreteModuleGenerator** (std::unique_ptr< BodyGenerator > bodyGenerator, std::unique_ptr< DefaultsGenerator > defaultsGenerator={})
- virtual void **generateBody** (InlineGenerator &gen, std::ostream &out) override
- virtual void **generateDefaults** (InlineGenerator &gen, std::ostream &out) override

7.4.1 Detailed Description

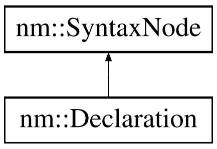
The ConcreteModuleGenerator class.

The documentation for this class was generated from the following file:

- codegeneration/concretemodulegenerator.hpp

7.5 nm::Declaration Struct Reference

Inheritance diagram for nm::Declaration:



Public Member Functions

- **Declaration** (SignalType t, std::string s)
- **Declaration** (SignalType t, std::unique_ptr< Variable > v)
- virtual void **gen** (InlineGenerator &g, std::ostream &out) override

Public Attributes

- SignalType **type**
- std::unique_ptr< Variable > **id**

The documentation for this struct was generated from the following file:

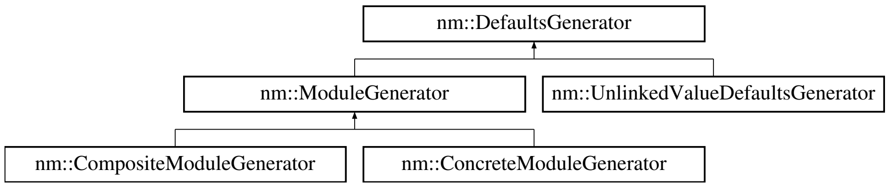
- codegeneration/inlinegenerator.hpp

7.6 nm::DefaultsGenerator Class Reference

Generates the definitions and default values module inputs.

```
#include <defaultsgenerator.hpp>
```

Inheritance diagram for nm::DefaultsGenerator:


nm::DefaultsGenerator
nm::ModuleGenerator
nm::UnlinkedValueDefaultsGenerator
nm::CompositeModuleGenerator
nm::ConcreteModuleGenerator

Public Member Functions

- virtual void **generateDefaults** (InlineGenerator &gen, std::ostream &out)=0

7.6.1 Detailed Description

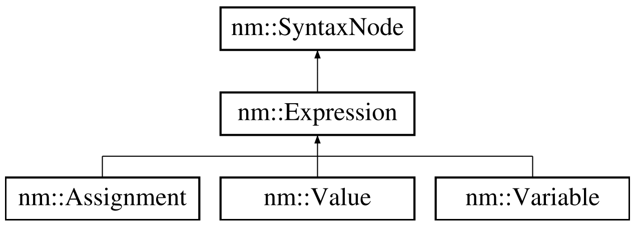
Generates the definitions and default values module inputs.

The documentation for this class was generated from the following file:

- codegeneration/defaultsgenerator.hpp

7.7 nm::Expression Struct Reference

Inheritance diagram for nm::Expression:



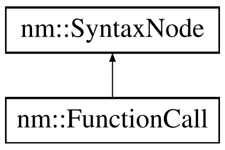
Additional Inherited Members

The documentation for this struct was generated from the following file:

- codegeneration/inlinegenerator.hpp

7.8 nm::FunctionCall Struct Reference

Inheritance diagram for nm::FunctionCall:



Public Member Functions

- template<typename T , typename U , typename V >
  FunctionCall (T &&function, U &&ins, V &&outs)
- virtual void **gen** (InlineGenerator &g, std::ostream &out) override

Public Attributes

- std::string **functionName**
- std::vector< Variable > **inputs**
- std::vector< Variable > **outputs**

The documentation for this struct was generated from the following file:

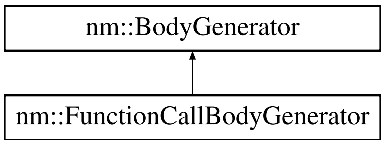
- codegeneration/inlinegenerator.hpp

7.9 nm::FunctionCallBodyGenerator Class Reference

The FunctionCallBodyGenerator class.

`#include <functioncallbodygenerator.hpp>`

Inheritance diagram for nm::FunctionCallBodyGenerator:



Public Member Functions

- template<typename T >
  FunctionCallBodyGenerator (T ∗∗t)
- virtual void **generateBody** (InlineGenerator &gen, std::ostream &out) override

7.9.1 Detailed Description

The FunctionCallBodyGenerator class.

The documentation for this class was generated from the following files:

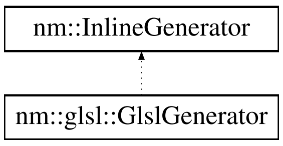
- codegeneration/functioncallbodygenerator.hpp
- codegeneration/functioncallbodygenerator.cpp

7.10 nm::glsl::GlslGenerator Class Reference

Generates glsl code to evaluate function graphs using OpenGL 3.0 shaders.

`#include <glslgenerator.hpp>`

Inheritance diagram for nm::glsl::GlslGenerator:



Static Public Member Functions

- static std::string **compileToGlslFunction** (const InputLink &inputLink, const OutputLink &outputLink, std↩ ::string name)
- static std::string **compileToGlslFunction** (std::vector< const InputLink ∗ > inputLinks, std::vector< const OutputLink ∗ > outputLink, std::string name)
- static std::string **compileToGlslFunctionWithDependencies** (const Module &module)
- static std::string **compileToGlslFunctionWithoutDependencies** (const Module &module)

Protected Member Functions

- virtual void **genTypeKeyword** (const SignalType &signalType, std::ostream &out) override
- virtual std::unique_ptr
  < nm::ModuleGenerator > **getModuleGenerator** (const Module &module) override

7.10.1 Detailed Description

Generates glsl code to evaluate function graphs using OpenGL 3.0 shaders.

The documentation for this class was generated from the following files:

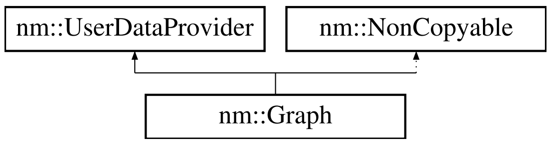
- codegeneration/glsl/glslgenerator.hpp
- codegeneration/glsl/glslgenerator.cpp

7.11 nm::Graph Class Reference

A graph of Modules.

`#include <graph.hpp>`

Inheritance diagram for nm::Graph:



Public Member Functions

- bool **addModule** (std::unique_ptr< Module > module)
- Module ∗ **createModule** (const ModuleType &type, std::string name)
- Module ∗ **createModule** (const ModuleType &type)
- std::unique_ptr< Module > **removeModule** (Module &module)
- void **clearModules** ()
- Module ∗ **getModule** (const std::string &name)
- const Module ∗ **getModule** (const std::string &name) const
- Module ∗ **getModule** (unsigned int index)
- const Module ∗ **getModule** (unsigned int index) const
- Module ∗ **findModule** (std::function< bool(Module &)> predicative)
- const Module ∗ **findModule** (std::function< bool(const Module &)> predicative) const
- unsigned int **numModules** () const
- void **traverseModulesTopological** (std::function< void(const Module &)> callback) const

Public Attributes

- signal< void(Graph &)> **destroying**
- signal< void(Graph &, Module &, unsigned int)> **moduleAdded**
- signal< void(Graph &, Module &, unsigned int)> **moduleRemoved**

7.11.1 Detailed Description

A graph of Modules.

The documentation for this class was generated from the following files:

- model/graph.hpp
- model/graph.cpp

7.12 nm::IdGenerator Class Reference

The IdGenerator class.

`#include <idgenerator.hpp>`

Public Member Functions

- **IdGenerator** (std::string prefix="nm_id_")
- std::string **getUniqueId** ()

7.12.1 Detailed Description

The IdGenerator class.

The documentation for this class was generated from the following file:

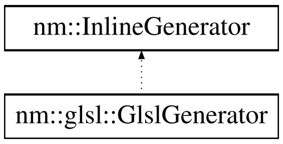
- codegeneration/idgenerator.hpp

7.13 nm::InlineGenerator Class Reference

Abstract base class for code generators.

`#include <inlinegenerator.hpp>`

Inheritance diagram for nm::InlineGenerator:



Classes

- struct InputRemap
- struct OutputRemap

Public Member Functions

- void **generateFromLinks** (const std::vector< InputRemap > &inputRemaps, const std::vector< Output↩ Remap > &outputRemaps, std::ostream &out)
- void **generateModule** (const Module &module, const std::vector< InputRemap > &inputRemaps, const std↩ ::vector< OutputRemap > &outputRemaps, std::ostream &out)
- std::string **getUniqueId** ()
- virtual std::unique_ptr
< ModuleGenerator > **getModuleGenerator** (const Module &module)

Protected Member Functions

- virtual void **genTypeKeyword** (const SignalType &signalType, std::ostream &out)=0
- virtual void **genDeclaration** (const Declaration &variable, std::ostream &out)
- virtual void **genAssignment** (const Assignment &assignment, std::ostream &out)
- virtual void **genVariable** (const Variable &variable, std::ostream &out)
- virtual void **genValue** (const SignalValue &value, std::ostream &out)
- virtual void **genFunctionCall** (FunctionCall &functionCall, std::ostream &out)

Friends

- struct **Assignment**
- struct **Variable**
- struct **Expression**
- struct **Value**
- struct **Declaration**
- struct **FunctionCall**

7.13.1 Detailed Description

Abstract base class for code generators.

The documentation for this class was generated from the following files:

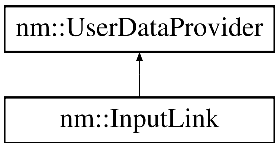
- codegeneration/inlinegenerator.hpp
- codegeneration/inlinegenerator.cpp

7.14 nm::InputLink Class Reference

Describes which output of which node a ModuleInput of a specific Module is connected to.

`#include <inputlink.hpp>`

Inheritance diagram for nm::InputLink:



Public Member Functions

- **InputLink** (Module &owner, const ModuleInput &type)
- bool **link** (OutputLink &output)
- void **unlink** ()
- const Module & **getOwner** () const
- Module & **getOwner** ()
- const ModuleInput & **getModuleInput** () const
- const OutputLink $*$ **getOutputLink** () const
- OutputLink $*$ **getOutputLink** ()
- SignalValue **getUnlinkedValue** () const
- bool **setUnlinkedValue** (SignalValue newValue)

Public Attributes

- signal< void(InputLink &)> **linkChanged**
- signal< void(InputLink &)> **unlinkedValueChanged**
- signal< void(InputLink &)> **destroying**

7.14.1 Detailed Description

Describes which output of which node a ModuleInput of a specific Module is connected to.

The documentation for this class was generated from the following files:

- model/inputlink.hpp
- model/inputlink.cpp

7.15 nm::InlineGenerator::InputRemap Struct Reference

Public Attributes

- std::string **externalName**
- const InputLink $*$ **inputLink**

The documentation for this struct was generated from the following file:

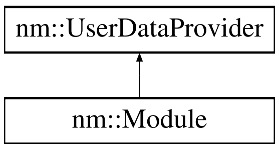
- codegeneration/inlinegenerator.hpp

7.16 nm::Module Class Reference

An instantiated ModuleType. A node in a function graph.

`#include <module.hpp>`

Inheritance diagram for nm::Module:



Public Member Functions

- Module (const ModuleType &type, std::string name, std::string description="")

  Constructor.
- const ModuleType & **getType** () const
- const ModuleType & **getType** ()
- const std::string **getName** () const
- void **setName** (std::string name)
- const std::string **getDescription** () const
- void **setDescription** (std::string description)
- InputLink $*$ **getInput** (std::string name)
- const InputLink $*$ **getInput** (std::string name) const
- InputLink $*$ **getInput** (unsigned int i)
- const InputLink $*$ **getInput** (unsigned int i) const
- unsigned int **getInputSize** () const
- std::vector$<$ InputLink $*$ $>$ **getInputs** ()
- OutputLink $*$ **getOutput** (std::string name)
- const OutputLink $*$ **getOutput** (std::string name) const
- OutputLink $*$ **getOutput** (unsigned int i)
- const OutputLink $*$ **getOutput** (unsigned int i) const
- unsigned int **getOutputSize** () const

- std::vector< OutputLink ∗ > **getOutputs** ()
- void disconnect ()

  disconnects module by unlinking all InputLinks and OutputLinks of this module.
- void **traverseChildren** (std::function< void(const Module &)> callback) const
- void **traverseParents** (std::function< void(const Module &)> callback) const
- void **traverseDescendants** (std::function< void(const Module &)> callback) const
- void **traverseDescendants** (std::function< void(Module &)> callback)
- void **traverseAncestors** (std::function< void(const Module &)> callback) const
- int getDepth () const
- int getHeight () const

Static Public Member Functions

- static std::vector< Module ∗ > **getDependenciesSorted** (const std::vector< OutputLink ∗ > &outputs, const std::set< InputLink ∗ > &ignoreInputs={})
- static std::vector< const Module ∗ > **getDependenciesSorted** (const std::vector< const OutputLink ∗ > &outputs, const std::set< const InputLink ∗ > &ignoreInputs={})
- static void **topologicallyTraverseDependencies** (const std::vector< OutputLink ∗ > &outputs, std↩ ::function< void(Module &)> visitor, const std::set< InputLink ∗ > &ignoreInputs={})
- static void **topologicallyTraverseDependencies** (const std::vector< const OutputLink ∗ > &outputs, std↩ ::function< void(const Module &)> visitor, const std::set< const InputLink ∗ > &ignoreInputs={})

Public Attributes

- signal< void(Module &, const std::string &)> nameChanged

  This signal is emitted when the name of the Module is changed.
- signal< void(Module &, const std::string &)> descriptionChanged

  This signal is emitted when the description of the Module is changed.
- signal< void(Module &)> destroying

  This signal is emitted before the Module is destroyed.
- signal< void(Module &, InputLink &)> addedInputLink

  This signal is emitted after an input has been added.
- signal< void(Module &, const ModuleInput &)> removedInputLink

  This signal is emitted after an input has been removed.
- signal< void(Module &, OutputLink &)> addedOutputLink

  This signal is emitted after an output has been added.
- signal< void(Module &, const ModuleOutput &)> removedOutputLink

  This signal is emitted after an output has been removed.
- signal< void(Module &)> dependenciesChanged

  This signal is emitted if changes have been made to the graph that may influence this Module's output signals.

7.16.1 Detailed Description

An instantiated ModuleType. A node in a function graph.

A module is a part of a node in a Graph. It has a corresponding ModuleType, which describes some sort of mathematical function or algorithm. A module may be thought of a configuration for a function call.

A Module has a number of InputLinks and OutputLinks that may be connected to other Modules InputLinks and OutputLinks.

Connections can be changed by using the getters for these InputLinks and OutputLinks. One InputLink exists for each ModuleInput in the ModuleType of the Module. The same applies for outputs.

Usually create by Graph::createModule

7.16.2 Constructor & Destructor Documentation

7.16.2.1 nm::Module::Module (const **ModuleType** & *type*, std::string *name*, std::string *description* = " ") `[explicit]`

Constructor.

Consider using Graph::createModule() instead if your goal is to add the module to a graph.

7.16.3 Member Function Documentation

7.16.3.1 int nm::Module::getDepth ( ) const

Returns

the number of modules above this one

7.16.3.2 int nm::Module::getHeight ( ) const

Returns

the number of modules below this one

The documentation for this class was generated from the following files:

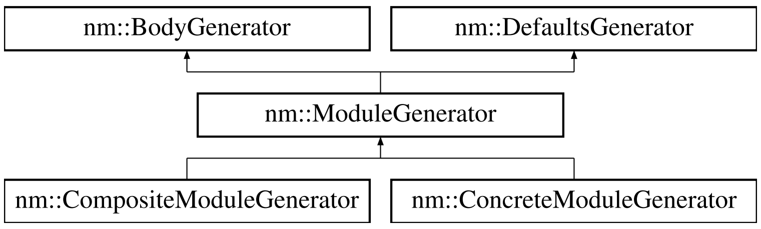
- model/module.hpp
- model/module.cpp

7.17 nm::ModuleGenerator Class Reference

The ModuleGenerator class.

`#include <modulegenerator.hpp>`

Inheritance diagram for nm::ModuleGenerator:



Additional Inherited Members

7.17.1 Detailed Description

The ModuleGenerator class.

The documentation for this class was generated from the following file:

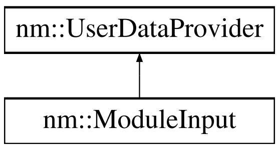
- codegeneration/modulegenerator.hpp

7.18 nm::ModuleInput Class Reference

Describes one of a ModuleType's inputs (name, SignalType, default value).

`#include <moduleinput.hpp>`

Inheritance diagram for nm::ModuleInput:



Public Member Functions

- **ModuleInput** (std::string name, SignalType signalType, const ModuleType &moduleType)
- **ModuleInput** (std::string name, SignalValue defaultValue, const ModuleType &moduleType)
- std::string **getName** () const
- SignalType **getSignalType** () const
- SignalValue **getDefaultValue** () const

Public Attributes

- signal< void(ModuleInput &)> **destroying**

7.18.1 Detailed Description

Describes one of a ModuleType's inputs (name, SignalType, default value).

The documentation for this class was generated from the following file:

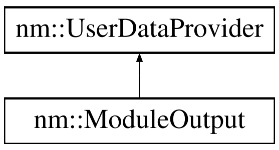
- model/moduleinput.hpp

7.19 nm::ModuleOutput Class Reference

Describes one of a ModuleType's outputs (name, SignalType)

`#include <moduleoutput.hpp>`

Inheritance diagram for nm::ModuleOutput:



Public Member Functions

- **ModuleOutput** (std::string name, SignalType signalType, const ModuleType &moduleType)
- std::string **getName** () const
- SignalType **getSignalType** () const

Public Attributes

- signal< void(ModuleOutput &)> **destroying**

7.19.1 Detailed Description

Describes one of a ModuleType's outputs (name, SignalType)

The documentation for this class was generated from the following file:

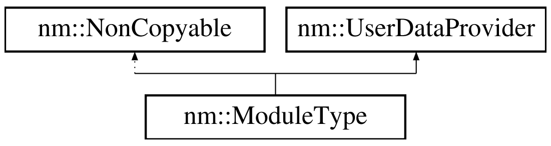
- model/moduleoutput.hpp

7.20 nm::ModuleType Class Reference

Describes a recipe for a module and its inputs and outputs.

`#include <moduletype.hpp>`

Inheritance diagram for nm::ModuleType:



Public Types

- enum Category { Category::Primitive, Category::Composite, Category::GraphInput, Category::GraphOutput }

  Module type category.

Public Member Functions

- ModuleType (std::string name, std::string description)

  Simple Constructor for composite module types.
- ModuleType (std::string name, Category category=Category::Composite, std::string description="")
- std::string getName () const

  A unique identifier for the module type.
- void **setName** (std::string name)
- std::string **getDescription** () const
- void **setDescription** (std::string description)
- bool isBuiltin () const

  A built-in module type is part of nmlib, and not created by the user.
- bool isComposite () const

  A composite module type is built from a graph of modules of other types.
- bool isPrimitive () const

  A primitive module type, is the opposite of a composite type.

- bool isRemovable () const

  Some modules, like the ones representing inputs and outputs, may not be removed from their graphs.
- bool isGraphInput () const

  A graph input module type represents the inputs of a composite module type.
- bool isGraphOutput () const

  A graph output module type represents the outputs of a composite module type.
- void setRemovable (bool removable)

  Change whether modules of this type may be removed from graphs or not.
- unsigned int **numInputs** () const
- const ModuleInput ∗ **getInput** (std::string name) const
- ModuleInput ∗ **getInput** (std::string name)
- const ModuleInput ∗ **getInput** (unsigned int index) const
- ModuleInput ∗ **getInput** (unsigned int index)
- void eachModuleInput (std::function< void(const ModuleInput &)> f) const

  Iterate over the inputs using the provided callback.
- void eachModuleInput (std::function< void(ModuleInput &)> f)

  Iterate over the inputs using the provided callback.
- ModuleInput ∗ addInput (std::string name, SignalType signalType)

  Add a new input to the module type.
- ModuleInput ∗ addInput (std::string name, SignalValue defaultValue)

  Add a new input to the module type.
- bool removeInput (ModuleInput ∗moduleInput)

  Removes an input from this module type.
- unsigned int **numOutputs** () const
- const ModuleOutput ∗ **getOutput** (std::string name) const
- ModuleOutput ∗ **getOutput** (std::string name)
- const ModuleOutput ∗ **getOutput** (unsigned int index) const
- ModuleOutput ∗ **getOutput** (unsigned int index)
- void eachModuleOutput (std::function< void(const ModuleOutput &)> f) const

  Iterate over the outputs using the provided callback.
- void eachModuleOutput (std::function< void(ModuleOutput &)> f)

  Iterate over the outputs using the provided callback.
- ModuleOutput ∗ addOutput (std::string name, SignalType signalType)

  Add a new output to the module type.
- bool removeOutput (ModuleOutput ∗moduleOutput)

  Removes an output from this module type.
- Graph ∗ getGraph ()

  Accessor for the graph of a composite module type.
- const Graph ∗ getGraph () const

  Accessor for the graph of a composite module type.
- Module ∗ getInputModule ()

  Accessor for the input module of a composite module type.
- const Module ∗ getInputModule () const

  Accessor for the input module of a composite module type.
- Module ∗ getOutputModule ()

  Accessor for the output module of a composite module type.
- const Module ∗ getOutputModule () const

  Accessor for the output module of a composite module type.
- ModuleOutput ∗ exportInternalOutput (OutputLink &outputLink, std::string externalName)

  Convenience method for exposing a part of the module graph as a new external output.

Public Attributes

- signal< void(ModuleType &, const std::string &)> nameChanged

  This signal is emitted when the name of the moduleType changes.
- signal< void(ModuleType &, const std::string &)> descriptionChanged

  This signal is emitted when the description of the moduleType changes.
- signal< void(ModuleInput &)> inputAdded

  This signal is emitted after an input has been added.
- signal< void(ModuleInput &)> removingInput

  This signal is emitted before an input is removed.
- signal< void(ModuleType &)> inputRemoved

  This signal is emitted after an input has been removed.
- signal< void(ModuleOutput &)> outputAdded

  This signal is emitted after an output has been added.
- signal< void(ModuleOutput &)> removingOutput

  This signal is emitted before an output is removed.
- signal< void(ModuleType &)> outputRemoved

  This signal is emitted after an output has been removed.
- signal< void(ModuleType &)> destroying

  This signal is emitted before the type is destroyed.

7.20.1 Detailed Description

Describes a recipe for a module and its inputs and outputs.

A module type may be thought of as a blueprint for a module. When a module is created, it is created according to a module type definition.

There are two main categories of module types, primitive and composite Composite module types, are built implemented as a graph of modules of other module types. Primitive module types are the lowest level building blocks of composite module types. They are defined by the library itself.

7.20.2 Member Enumeration Documentation

7.20.2.1 enum **nm::ModuleType::Category** `[strong]`

Module type category.

Enumerator

Primitive	A low-level module type, without a graph
Composite	A high-level module type, created as a composition of modules in a graph
GraphInput	A special module type for inputs of a graph
GraphOutput	A special module type for outputs of a graph

7.20.3 Constructor & Destructor Documentation

7.20.3.1 nm::ModuleType::ModuleType (std::string *name*, std::string *description*) `[explicit]`

Simple Constructor for composite module types.

Parameters

name	A unique identifier for the module type
description	A comment or description of what kind of function the module type represents.

7.20.3.2 nm::ModuleType::ModuleType (std::string *name*, ModuleType::Category *category* = Category::Composite, std::string *description* = "") [explicit]

Parameters

name	A unique identifier for the module type
category	Whether to create a composite, primitive, or special module type
description	A comment or description of what kind of function the module type represents.

7.20.4 Member Function Documentation

7.20.4.1 ModuleInput ∗ nm::ModuleType::addInput (std::string *name*, SignalType *signalType*)

Add a new input to the module type.

Parameters

name	The name to give the new input
signalType	The type of the new input

The default value for this input is set to zero

7.20.4.2 ModuleInput ∗ nm::ModuleType::addInput (std::string *name*, SignalValue *defaultValue*)

Add a new input to the module type.

Parameters

name	The name to give the new input
defaultValue	The default value for the new input

The type of the new input is inferred from the value

7.20.4.3 ModuleOutput ∗ nm::ModuleType::addOutput (std::string *name*, SignalType *signalType*)

Add a new output to the module type.

Parameters

name	The name to give the new output
signalType	The type of the new output

The default value for this output is set to zero

7.20.4.4 ModuleOutput ∗ nm::ModuleType::exportInternalOutput (OutputLink & *outputLink*, std::string *externalName*)

Convenience method for exposing a part of the module graph as a new external output.

This method only works for composite module types.

7.20.4.5 Graph∗ nm::ModuleType::getGraph ( ) [inline]

Accessor for the graph of a composite module type.

Returns

Returns a pointer to the Graph corresponding to this module type, or nullptr if this is not a composite module type

7.20.4.6 const Graph∗ nm::ModuleType::getGraph ( ) const [inline]

Accessor for the graph of a composite module type.

Returns

Returns a pointer to the Graph corresponding to this module type, or nullptr if this is not a composite module type

7.20.4.7 const Module ∗ nm::ModuleType::getInputModule ( ) const

Accessor for the input module of a composite module type.

7.20.4.8 const Module ∗ nm::ModuleType::getOutputModule ( ) const

Accessor for the output module of a composite module type.

7.20.4.9 bool nm::ModuleType::isRemovable ( ) const [inline]

Some modules, like the ones representing inputs and outputs, may not be removed from their graphs.

Returns

Whether modules of this type may be removed from their graphs

7.20.4.10 bool nm::ModuleType::removeInput (ModuleInput ∗ *moduleInput*)

Removes an input from this module type.

Parameters

moduleInput	A pointer to one of this ModuleType's inputs

Returns

true if an input was removed

Instantiated Modules of this type are notified of this, and their corresponding InputLinks are automatically removed.

7.20.4.11 bool nm::ModuleType::removeOutput (ModuleOutput ∗ *moduleOutput*)

Removes an output from this module type.

Parameters

moduleOutput	A pointer to one of this ModuleType's outputs

Returns

true if an output was removed

Instantiated Modules of this type are notified of this, and their corresponding OutputLinks are automatically removed.

The documentation for this class was generated from the following files:

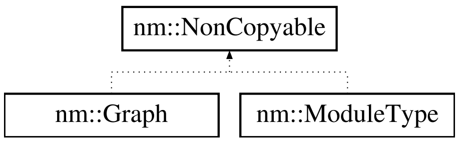
- model/moduletype.hpp
- model/moduletype.cpp

7.21 nm::NonCopyable Class Reference

A super-class for non-copyable classes.

`#include <noncopyable.hpp>`

Inheritance diagram for nm::NonCopyable:



7.21.1 Detailed Description

A super-class for non-copyable classes.

The documentation for this class was generated from the following file:

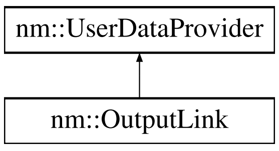
- util/noncopyable.hpp

7.22 nm::OutputLink Class Reference

an output of a Module

`#include <outputlink.hpp>`

Inheritance diagram for nm::OutputLink:



Public Member Functions

- OutputLink (Module &owner, const ModuleOutput &type)

  OutputLink.
- bool addLink (InputLink &input)

  Add a connection from this OutputLink to the specified InputLink.
- bool unlink (InputLink ∗input)

  Remove the link an InputLink if it exists.
- void unlinkAll ()

  Remove links to all inputs.
- unsigned int **numLinks** ()
- InputLink ∗ **getLink** (unsigned int index)
- const Module & **getOwner** () const
- Module & **getOwner** ()
- const ModuleOutput & **getModuleOutput** () const

Public Attributes

- signal< void(OutputLink &)> **linksChanged**
- signal< void(OutputLink &)> **destroying**

7.22.1 Detailed Description

an output of a Module

OutputLinks are created by the library itself when instantiation a new Module

7.22.2 Constructor & Destructor Documentation

7.22.2.1 nm::OutputLink::OutputLink (**Module** & *owner,* const **ModuleOutput** & *type*) `[inline]`, `[explicit]`

OutputLink.

Parameters

owner	which module this is an output of
type	which ModuleOutput this is an outputLink for

7.22.3 Member Function Documentation

7.22.3.1 bool nm::OutputLink::addLink (**InputLink** & *input*)

Add a connection from this OutputLink to the specified InputLink.

Parameters

input	

Returns

Whether a new link was added.

7.22.3.2 bool nm::OutputLink::unlink (**InputLink** $*$ *input*)

Remove the link an InputLink if it exists.

Parameters

input	

Returns

Whether a link was removed

7.22.3.3 void nm::OutputLink::unlinkAll ( )

Remove links to all inputs.

This also removes the links from the inputs to the output

The documentation for this class was generated from the following files:

- model/outputlink.hpp
- model/outputlink.cpp

7.23 nm::InlineGenerator::OutputRemap Struct Reference

Public Attributes

- std::string **externalName**

- const OutputLink * **outputLink**

The documentation for this struct was generated from the following file:

- codegeneration/inlinegenerator.hpp

7.24 nm::Parser Class Reference

Converts json strings to TypeManagers.

```
#include <parser.hpp>
```

Public Member Functions

- optional< std::unique_ptr
  < TypeManager > > **parseDocument** (std::string json)

7.24.1 Detailed Description

Converts json strings to TypeManagers.

A parser is typically used like this:

```
//replace readfile with your file reading function
std::string json = readFile("terrain.nm.json");
Parser parser;
auto maybeTypeManager = parser.parseDocument(json);
if(!maybeTypeManager){
    //error handling
}
TypeManager& typeManager = **maybeTypeManager;

//use the type manager
```

The documentation for this class was generated from the following files:

- serialization/parser.hpp
- serialization/parser.cpp

7.25 nm::Serializer Class Reference

Serializes a TypeManager to a JSON string.

```
#include <serializer.hpp>
```

Public Member Functions

- std::string **serialize** (const TypeManager &typeManager)

7.25.1 Detailed Description

Serializes a TypeManager to a JSON string.

The documentation for this class was generated from the following files:

- serialization/serializer.hpp
- serialization/serializer.cpp

7.26 nm::SignalType Class Reference

Describes a the dimensionality of a signal Can be extended to include other type information such as distinctions between doubles, flots and ints as well.

`#include <signaltype.hpp>`

Public Member Functions

- **SignalType** (int dimensions)
- bool **operator==** (const SignalType &rhs) const
- bool **operator!=** (const SignalType &rhs) const
- bool isConvertibleTo (const SignalType &rhs) const

  Checks if this SignalType can be converted to another.

Public Attributes

- const int **dimensionality**

7.26.1 Detailed Description

Describes a the dimensionality of a signal Can be extended to include other type information such as distinctions between doubles, flots and ints as well.

7.26.2 Member Function Documentation

7.26.2.1 bool nm::SignalType::isConvertibleTo (const SignalType & *rhs*) const

Checks if this SignalType can be converted to another.

Parameters

SignalType	to convert to

Returns

The documentation for this class was generated from the following files:

- model/signaltype.hpp
- model/signaltype.cpp

7.27 nm::SignalValue Class Reference

A vector of floats that can be set as the unlinked value of InputLinks.

`#include <signalvalue.hpp>`

Public Member Functions

- **SignalValue** (float value)
- **SignalValue** (std::vector< float > values)
- **SignalValue** (SignalType signalType)
- **SignalValue** (const SignalValue &other)

- SignalValue & **operator=** (const SignalValue &rhs)
- float & **operator[]** (unsigned int i)
- float **operator[]** (unsigned int i) const
- SignalType **getSignalType** () const

7.27.1 Detailed Description

A vector of floats that can be set as the unlinked value of InputLinks.

The documentation for this class was generated from the following file:

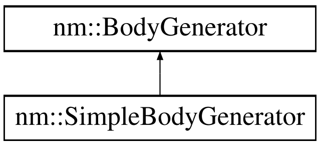
- model/signalvalue.hpp

7.28 nm::SimpleBodyGenerator Class Reference

The SimpleBodyGenerator class.

```
#include <simplebodygenerator.hpp>
```

Inheritance diagram for nm::SimpleBodyGenerator:



Public Member Functions

- **SimpleBodyGenerator** (std::string body)
- virtual void **generateBody** (InlineGenerator &, std::ostream &out)

7.28.1 Detailed Description

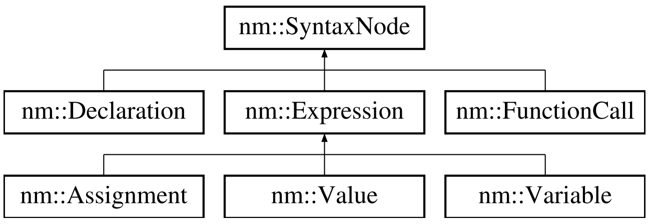
The SimpleBodyGenerator class.

The documentation for this class was generated from the following file:

- codegeneration/simplebodygenerator.hpp

7.29 nm::SyntaxNode Struct Reference

Inheritance diagram for nm::SyntaxNode:



Public Member Functions

- virtual void **gen** (InlineGenerator &gen, std::ostream &out)=0

The documentation for this struct was generated from the following file:

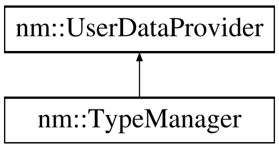
- codegeneration/inlinegenerator.hpp

7.30 nm::TypeManager Class Reference

The top-level entity of a noise model. Encapsulates several user types as well as built-in types.

`#include <typemanager.hpp>`

Inheritance diagram for nm::TypeManager:



Public Member Functions

- TypeManager ()

    Creates a new TypeManager.
- bool addUserType (std::unique_ptr< ModuleType > moduleType)

    Add an existing user type to the TypeManager.
- ModuleType ∗ createUserType (std::string desiredName)

    try to create a new usertype with the given name
- const ModuleType ∗ getType (std::string name) const

    Search for a module type with the given name.
- const ModuleType ∗ getBuiltinType (std::string name) const

    Search for a built-in type with the given name.
- const ModuleType ∗ getUserType (std::string name) const

    Search for a user type with the given name.
- ModuleType ∗ getUserType (std::string name)

    Search for a user type with the given name.
- void initBuiltinTypes ()

    Populate the list of built-in module types.
- unsigned int **numBuiltinTypes** () const
- const ModuleType ∗ **getBuiltinType** (unsigned int index) const
- unsigned int **numUserTypes** () const
- const ModuleType ∗ **getUserType** (unsigned int index) const

Public Attributes

- signal< void(TypeManager &)> destroying

    This signal is emitted before the TypeManager is destroyed.
- signal< void(TypeManager &)> userTypesChanged

    This signal is emitted when the list of user type changes.

7.30.1 Detailed Description

The top-level entity of a noise model. Encapsulates several user types as well as built-in types.

A type manager is the top-level entity in the model hierarchy.

Two lists of ModuleTypes are maintained in a TypeManager, the user types, and the built-in types.

A user type may be constructed explicitly, or may be returned from a Parser in the form of a unique_ptr.

New user types can conveniently be added to the type manager, using the method createUserType().

A TypeManager owns the ModuleTypes it manages.

7.30.2 Constructor & Destructor Documentation

7.30.2.1 nm::TypeManager::TypeManager ( )

Creates a new TypeManager.

Note that initBuiltinTypes method has to be called explicitly to populate the list of buil-in types.

7.30.3 Member Function Documentation

7.30.3.1 bool nm::TypeManager::addUserType (std::unique_ptr< **ModuleType** > *moduleType*)

Add an existing user type to the TypeManager.

Returns

true if there was not a naming conflict and the type was added.

To create a new user type, consider using createUserType instead.

7.30.3.2 **ModuleType** ∗ nm::TypeManager::createUserType (std::string *desiredName*)

try to create a new usertype with the given name

Returns

If a ModuleType was created, a pointer to it is returned. Note that the ModuleType is still owned by the TypeManager.

If the name is unavailable a different name may be chosen.

7.30.3.3 const **ModuleType** ∗ nm::TypeManager::getType (std::string *name*) const

Search for a module type with the given name.

Returns

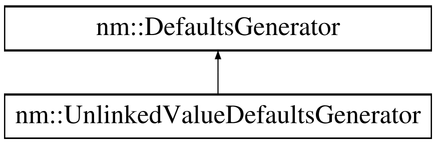
A pointer to a user type or built-in type, or nullptr if it was not found.

The documentation for this class was generated from the following files:

- model/typemanager.hpp
- model/typemanager.cpp

7.31 nm::UnlinkedValueDefaultsGenerator Class Reference

Inheritance diagram for nm::UnlinkedValueDefaultsGenerator:



Public Member Functions

- **UnlinkedValueDefaultsGenerator** (const Module &module)
- virtual void **generateDefaults** (InlineGenerator &gen, std::ostream &out) override

The documentation for this class was generated from the following files:

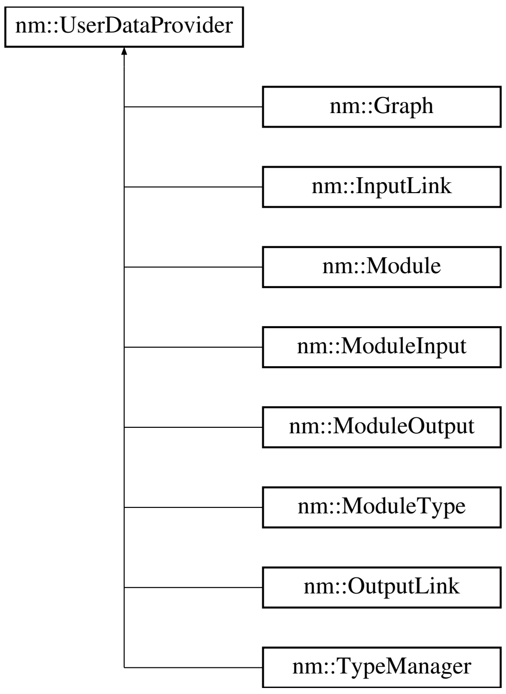
- codegeneration/unlinkedvaluedefaultsgenerator.hpp
- codegeneration/unlinkedvaluedefaultsgenerator.cpp

7.32 nm::UserDataProvider Class Reference

Base class for stuff that needs to provide user data in form of a void∗ pointer.

`#include <userdataprovider.hpp>`

Inheritance diagram for nm::UserDataProvider:



Public Member Functions

- void ∗ **getUserData** () const
- void **setUserData** (void ∗userData)

7.32.1 Detailed Description

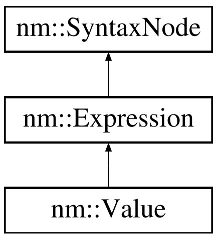
Base class for stuff that needs to provide user data in form of a void∗ pointer.

The documentation for this class was generated from the following file:

- util/userdataprovider.hpp

7.33 nm::Value Struct Reference

Inheritance diagram for nm::Value:



Public Member Functions

- **Value** (std::unique_ptr< SignalValue > v)
- virtual void **gen** (InlineGenerator &g, std::ostream &out) override

Public Attributes

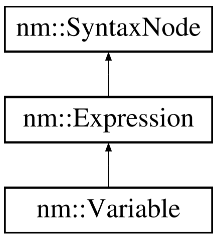
- std::unique_ptr< SignalValue > **value**

The documentation for this struct was generated from the following file:

- codegeneration/inlinegenerator.hpp

7.34 nm::Variable Struct Reference

Inheritance diagram for nm::Variable:



Public Member Functions

- **Variable** (std::string id)
- virtual void **gen** (InlineGenerator &g, std::ostream &out) override

Public Attributes

- std::string **m_id**

The documentation for this struct was generated from the following file:

- codegeneration/inlinegenerator.hpp

8 File Documentation

8.1 model.hpp File Reference

The model module.

```
#include <nmlib/model/module.hpp>
#include <nmlib/model/moduletype.hpp>
#include <nmlib/model/moduleinput.hpp>
#include <nmlib/model/moduleoutput.hpp>
#include <nmlib/model/inputlink.hpp>
#include <nmlib/model/outputlink.hpp>
#include <nmlib/model/signaltype.hpp>
#include <nmlib/model/typemanager.hpp>
#include <nmlib/model/graph.hpp>
```

8.1.1 Detailed Description

The model module.

8.2 serialization.hpp File Reference

The serialization module.

```
#include <nmlib/serialization/parser.hpp>
#include <nmlib/serialization/serializer.hpp>
```

8.2.1 Detailed Description

The serialization module.

8.3 util.hpp File Reference

Utility headers.

```
#include <nmlib/util/makeunique.hpp>
#include <nmlib/util/noncopyable.hpp>
#include <nmlib/util/userdataprovider.hpp>
```

8.3.1 Detailed Description

Utility headers.

Index