

Noise Modeler User's Guide

1 Introduction

Noise Modeler is an application for designing noise-based functions for generation of height-map 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.

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.

2 Installation

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.

2.2 Prebuilt binaries

An installer and a standalone binary are available for Windows at <http://johan.helsing.priv.no/noisemodeler>.

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

3 Building from source

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

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)

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 ([Section 0.3](#)).

Arch Linux users may use the following command:

```
1 $ sudo pacman -S qt5-base qt5-svg qt5-tools \  
2 qt5-declarative qt5-quickcontrols \  
3 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Ubuntu It is easiest if you have version 14.04 or newer, as earlier versions only have outdated development packages in the official repositories.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:

```
1 $ sudo apt-get install qt-sdk libqt5svg5-dev gcc \
2 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:

```
1 $ 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:

```
1 $ mkdir build-noisemodeler
2 $ 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.

```
1 $ qmake ../noisemodeler
```

Compile the project:

```
1 $ 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.

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:

```
1 $ 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).

4 Tutorial

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

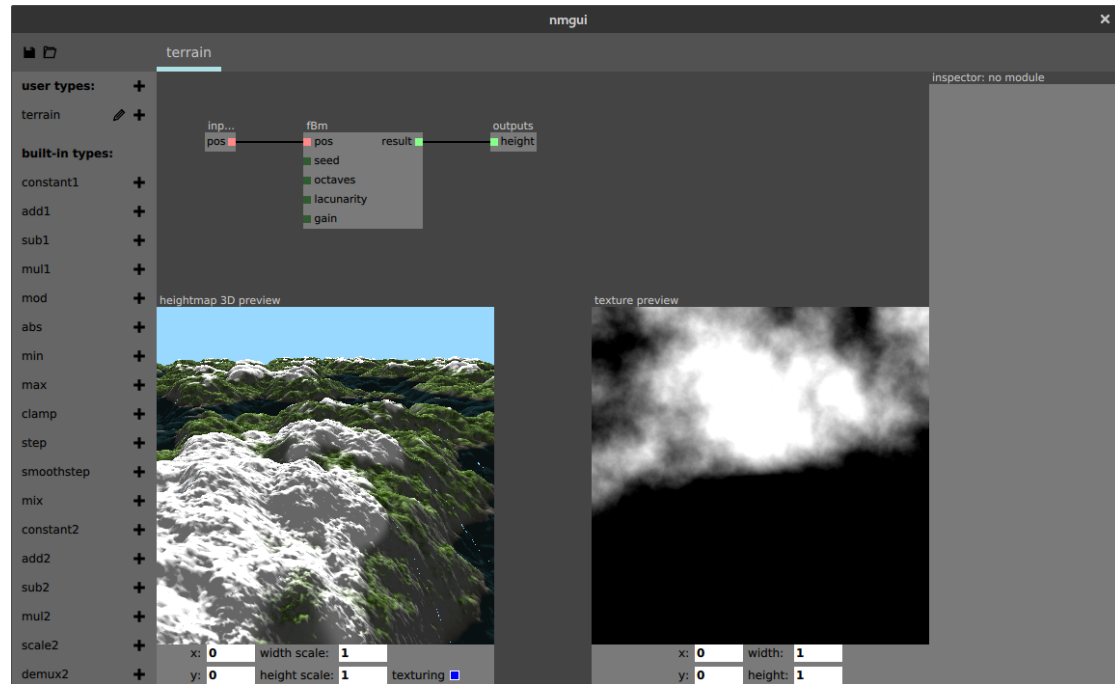


Figure 1: Screenshot of the application right after it has been started

When you first start the application, you will be greeted with the interface shown in Fig. 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 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 landscape moving quickly around. All text boxes in the application behave in a similar way, 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.

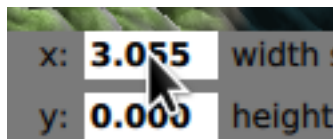


Figure 2: Use your mouse scroll wheel while hovering over text boxes to easily adjust values.

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 longitude?”

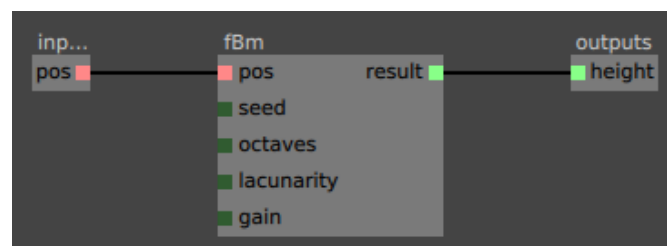


Figure 3: The default module graph.

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.

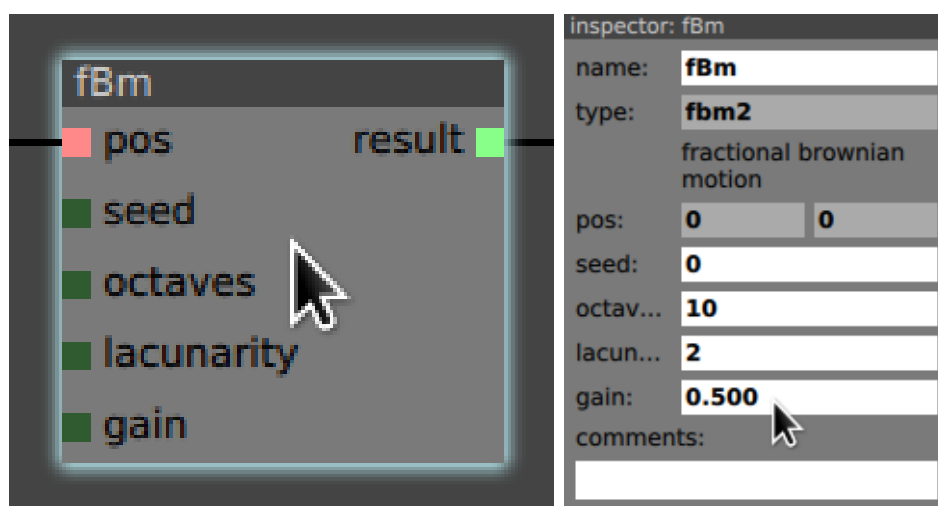


Figure 4: Click the fBm module so it can be edited in the inspector.

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, let's try to change some of the inputs to 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. 5 happening. Set the gain back to 0.5 before you continue.

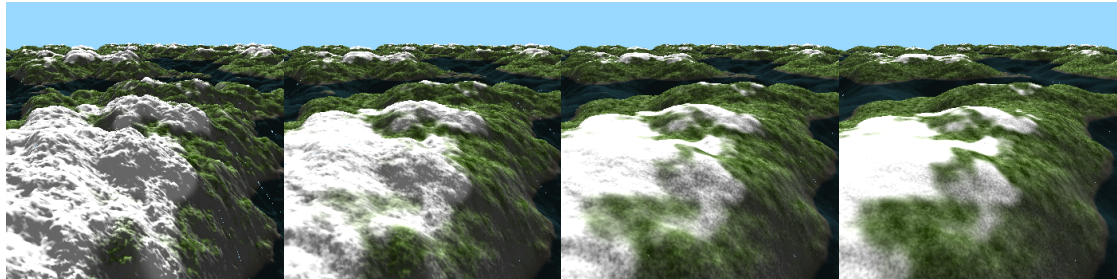


Figure 5: Adjusting the gain of fBm

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 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 your 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 terrain 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.

5 User interface


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

5.1 Saving documents

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

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 can be achieved through the command line, by entering:

```
1 $ /path/to/nmgui /path/to/myterrain.nm.json
```

This may also be achieved by dragging a document on top of the executable or the icon that starts the application.

5.3 Tabs

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.

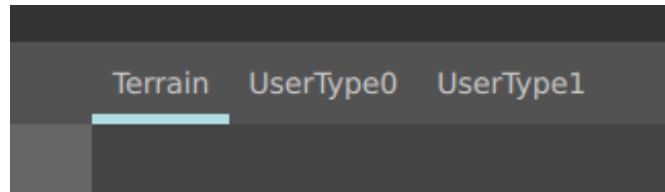


Figure 6: Row of open tabs. The current tab, “Terrain”, is highlighted.

5.4 Module type list

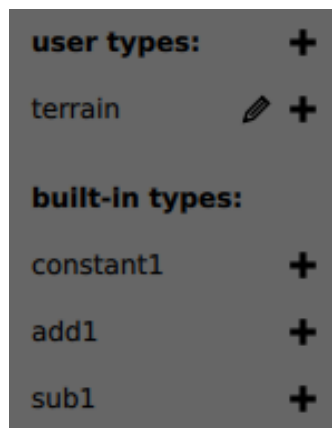



Figure 7: Module type list

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 be 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.

5.5 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 ([Section 0.5](#)).

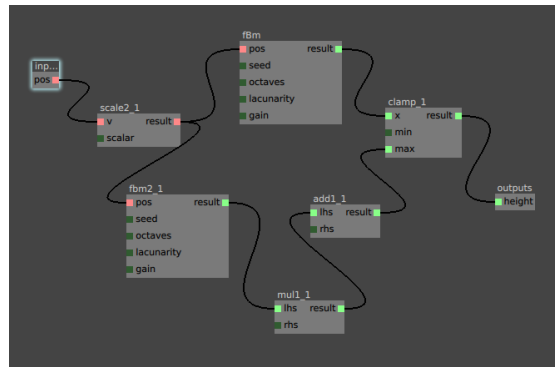


Figure 8: The graph editor

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 too 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.

5.6 2D preview

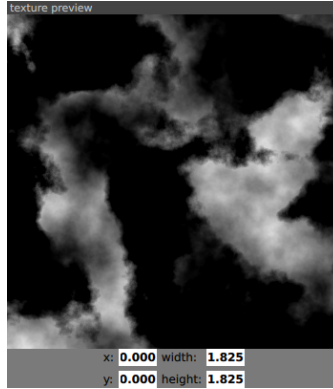


Figure 9: 2D texture preview of the terrain.

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.

5.7 3D preview

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

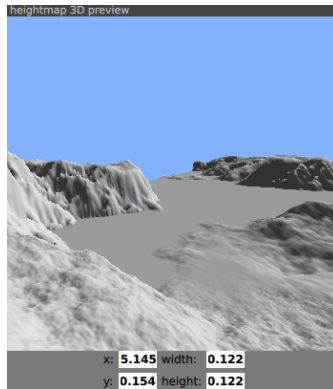


Figure 10: 3D preview of the terrain.

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.

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.

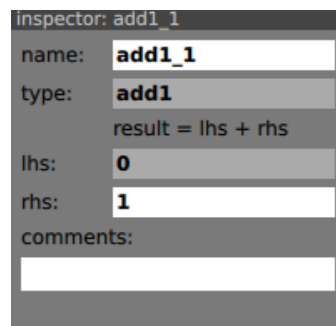


Figure 11: The inspector showing an add1 module, a one-dimensional addition module. Only the “rhs” (right-hand-side) value is editable. “lhs” is gray and cannot be edited because it is currently connected to the output of another module.

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.

6 Module types

Table 1 contains a list of the available module types, as well as a mathematical definition for their behaviour. 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 [1].

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.

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.

```
1 std::string serializedGraph = readFileToString("terrain.nm.json"↵
    );
```

Then parse the json into a TypeManager.

```
1 nm::Parser parser;
2 nm::optional<std::unique_ptr<nm::Parser>> maybeTypeManager = ↵
    parser.parse(serializedGraph);
3
4 //check if parsing succeeded
5 if(!maybeTypeManager){
6     //TODO handle errors
7     exit(EXIT_FAILURE);
8 }
9
10 //create an alias to the typeManager for more convenient access
```

```
11 nm::TypeManager &typeManager = *(*maybeTypeManager);
```

After this, you may use the `typeManager` alias to access any “user types” defined in the file.

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:

```
1 //we will be using the nm namespace frequently
2 using namespace nm;
3
4 //get the relevant graph
5 ModuleType* terrainModuleType = typeManager.getUserType("terrain↵
6 ");
7
8 //get the graph of the module
9 Graph* graph = terrainModuleType->getGraph();
10
11 //get the input and output for your function
12 InputLink* input = graph->getModule("inputs")->getInput("pos");
13 OutputLink* output = graph->getModule("outputs")->getOutput("↵
14 height");
15
16 //generate a function called "elevation".
17 std::string glslSourceCode = glsl::GlslGenerator::↵
18     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:

```
1 vec2 pos = vertexPosition.xy;
2 float height;
3 elevation(pos, height);
4 gl_Position = vec4(vertexPosition.xy, height, 1);
```


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> | $lhs(D), rhs(D)$ | $result(D)$ | $result = lhs + rhs$ |
| sub<D> | $lhs(D), rhs(D)$ | $result(D)$ | $result = lhs - rhs$ |
| mul<D> | $lhs(D), rhs(D)$ | $result(D)$ | $result = lhs \cdot rhs$ |
| scale<D> | $v(D), scalar$ | $result(D)$ | $result = scalar \cdot v$ |
| mod | $dividend, divisor$ | $result$ | $result \equiv dividend \bmod divisor$ |
| min | a, b | $result$ | $result = \begin{cases} a & \text{if } a < b \\ b & \text{if } a \geq b \end{cases}$ |
| max | a, b | $result$ | $result = \begin{cases} a & \text{if } a \geq b \\ b & \text{if } a < b \end{cases}$ |
| abs | $source$ | $result$ | $result = source $ |
| clamp | x, a, b | $result$ | $result = \begin{cases} x & \text{if } a < x < b \\ a & \text{if } x \leq a \\ b & \text{if } x \geq b \end{cases}$ |
| step | $value, edge$ | $result$ | $result = \begin{cases} 0 & \text{if } value < edge \\ 1 & \text{if } value \geq edge \end{cases}$ |
| 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 | $result$ | $result = smoothstep(x, 0, 1) \cdot a + smoothstep(x, 1, 0) \cdot b$ |
| 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 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.