UPPSALA
UNIVERSITET

IT 15045
Examensarbete 30 hp
Juni 2015

Interactive Methods for Procedural Texture Generation with Noise

Boris Kachscovsky

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Interactive Methods for Procedural Texture Generation with Noise

Boris Kachscovsky

Many computer graphics applications have used procedural noise since the 1980s, but there are still very few tools which allow non programming-oriented users to make procedural textures. This paper attempts to provide a framework for building and creating procedural noise based textures, in a way that can be easily abstracted and understood by those users. A careful study is conducted of Perlin Noise, and similar interfaces and tools are examined in order to create a framework centered around composable parts and semantic abstractions. The framework is then used to build a proof-of-concept interface which exemplifies some of the conclusions drawn from the study. The proof-of-concept interface successfully creates an environment which can be used to create procedural textures, and serves as a guide for future interfaces in the field.

Handledare: Martin Strandgren
Ämnesgranskare: Anders Hast
Examinator: Edith Ngai
IT 15045
Tryckt av: Reprocentralen ITC

Contents

Chapter 1

Introduction

1.1 Background

Procedural noise has grown considerably in popularity since it's inception. Many modern tools make use of its advantages, namely its speed, low memory footprint, and general ability to create complex visual detail with very little overhead. Today its main advantage is the ability for procedural noise to be arbitrarily accessible, allowing for easy integration with modern GPUs [LLC+10].

Yet for all its use, procedural noise is still a daunting and unapproachable tool for most users without a programming background. This is partially owing to the inherent "pseudo-random" nature of noise algorithms, which are usually expressed statistically and mathematically rather than visually. It can also be attributed to the complexity of modern day graphics programming, which requires an increasing understanding of both the hardware *and* the result. Thus there are very few tools these users have at their disposal [LLC+10].

That said, new tools and programs now exist which allow users to create 3D scenes with little to no programming required. Whats more, these new tools provide abstractions which allow users to better understand and work with once difficult concepts such as lighting and illumination with ease. Procedural noise has not followed suit.

This paper intends to define a framework which works as an arbiter between the current standard practices of working with noise, and the user who is not as programming centered. In essence, I intend to create a design paradigm which fosters users' creativity and forces them to think procedurally. This will allow users who may have found noise inaccessible in the past to discover it anew, which could in turn create whole new uses for noise that had not been previously imagined.

1.2 Motivation

Procedural texturing tools are essential to modern day 3D graphics suites. Nevertheless, current methods for working with procedural textures are either too rigid to allow for exploration or are too complex for those who are not already experienced programmers.

A balance can be achieved through composition oriented design. This means that the design should focus on creating small well understood parts which can be easily abstracted. This in turn allows the user to develop their own personal abstractions by piecing these parts together, ultimately creating an N-Dimensional procedural texture.

Noise is the focal point of this analysis of 3D texture interface design precisely because of the difficulty it presents. Because noise is meant to be stochastic, it *requires* powerful abstractions in order for it to be meaningful, else it is simply "random". Given this issue, the resultant design will necessitate a clear and composable style. The design framework's goal is not to *teach* new skills, but instead to empower users to make connections themselves.

1.3 Approach

The study will be conducted in five distinct parts:

1. An analysis of related work.
2. A description and definition of noise and procedural noise, as well as an analysis of one specific type of procedural noise, namely Perlin Noise and its common applications.
3. A description of a framework for a procedural texturing interface.
4. A prototype implementation of a texture editing which conforms to the framework.
5. A discussion of my results.

By looking into one specific type of noise and building a framework around its common use, the essential building blocks of the interface can be found. They can then abstracted upon, resulting in a framework which achieves the balance between rigidity and complexity.

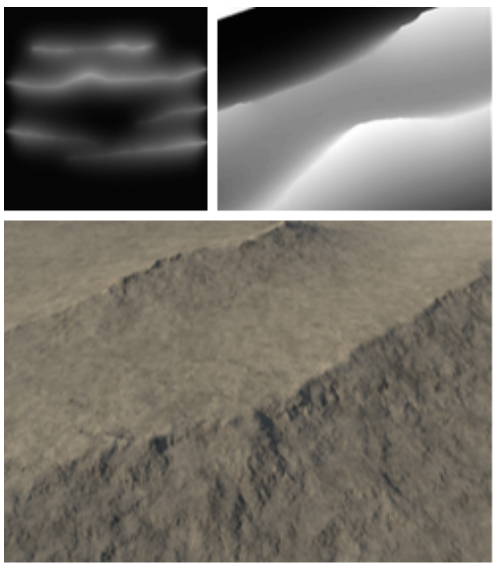
The implementation and discussion will serve as a proof-of-concept in order to give an example of the framework and hopefully pave the way for future creativity-oriented designs.

Chapter 2

Related Work

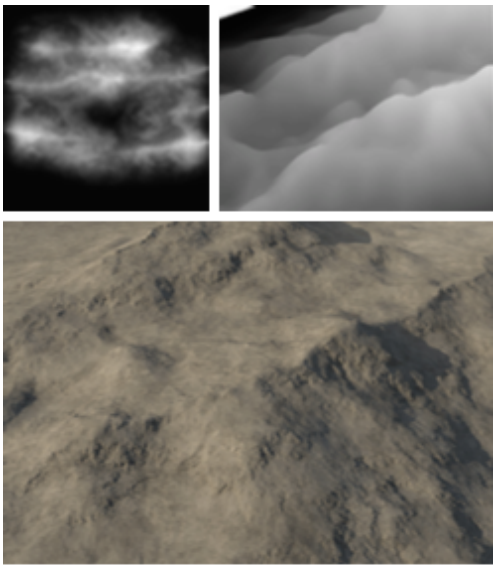
Below are a few examples of work in the same general field of creating interfaces for working with noise, function composition, and procedural textures. While each of the interfaces below are meant for extremely different purposes - they all share a common theme of building complex functions to be used in computer graphics. What is of particular concern for the purposes of this paper is the method by which they communicate the concepts the interfaces are attempting to grapple, with the user.

2.1 Frequency Based Controls



$\alpha_{high} = 0, \alpha_{low} = 0$

(a) Low frequency and high frequency fractal detail shown here at a lower setting.



$\alpha_{high} = 1, \alpha_{low} = 1$

(b) Low frequency and high frequency fractal detail shown here at a higher setting.

Figure 2.1: Above are two images which show some of the results of the interface with the interactive components that can be used to add noise. On the left of each image is its elevation map, on the right is its 3D interface view, and on the bottom is the resulting rendered 3D terrain [BCA+14].

Bradbury et. al.'s frequency based controls for terrain editing crafted an interesting solution to the user interface challenge of designing with noise. They propose a method which can have plausible results while maintaining an intuitive control by focusing on how landscapes are drawn on paper.

The system requires two inputs, one is a height map used for reference, and the other is a user provided ridge map. The ridge map is produced by drawing mountains on a canvas from the first person perspective (i.e. a typical landscape painting). Then, on the same canvas, the user adjusts a perspective grid in order to put the 2D mountains into a 3D context. A height map is then produced that approximately matches the drawing made by the user. The resulting height map can then be manipulated via common image processing operations such as copy and paste. Each resulting edit produces a new layer, allowing each layer to be merged or deleted. Once the height map is produced, noise can be added by using the frequencies on the referenced height map and adding them to the ridge height map [BCA+14].

While this is not an example of procedural noise, it is an interesting idea for an interface which combines the idea of landscape composition and terrain design. In that sense, the interface is an attempt to bridge the gap between the artistic and computational domains in a way that does not sacrifice functionality. However, the requirement that the noise must be based on reference material limits the interface. As this method is meant for only for terrain generation which mimics reference material, it works very well - but a Procedural noise method requires much more direct control of the noise.

2.2 Gabor Noise Interface

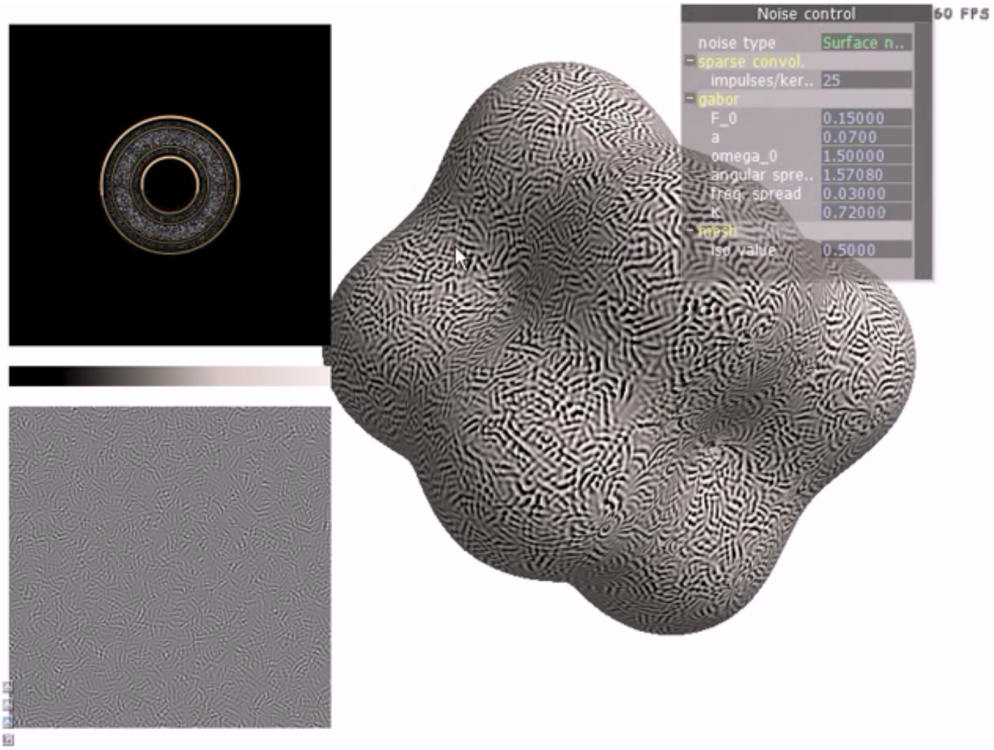


Figure 2.2: Above is the Gabor Noise interface. It contains the spectral representation of the kernel (top left), the resultant noise represented in 2D (bottom left) and the color bar the noise maps to in between. The figure in the center shows the resultant colors mapped onto the object [LLDD09].

The Gabor Noise interface is an impressive attempt to redefine the most common ways in which noise

interfaces are typically used. It centers the entire design on direct manipulation in the spectral domain, thereby focusing solely on creating noise, and not on using noise to interpolate or displace functions (as in the previous example).

The crux of the design is centered around using the Gabor kernel with a sparse convolution process. The Gabor Kernal is manipulated in the spectral domain, and once the desired spectral qualities have been found, the spectral image is converted into the spacial domain convulved with sparse convolution process.[1] The Gabor kernel is used because it has compact support both in the spacial and the frequency domains [LLDD09]. This results in a small section of a pattern. That pattern is spread across an N-Dimensional space by some user defined input on, for example, the pattern's spread and rotation.

This is a particularly interesting approach as it is one of the few interfaces that center wholly around noise and noise interface design. Its concentration on how the noise is produced allows users to get much closer to the resultant noise than, say, the former interface which only had two "slider" values for noise (low and high frequencies).

That said, the abstraction presented with the manipulation in the spectral domain does not necessarily hold a strong connection to the resulting shape. Users will need to have a strong intuitive understanding of Fourier transforms in order to work with the interface. Without it, it becomes very difficult to couple the manipulated variables with the result.

2.3 Shadershop

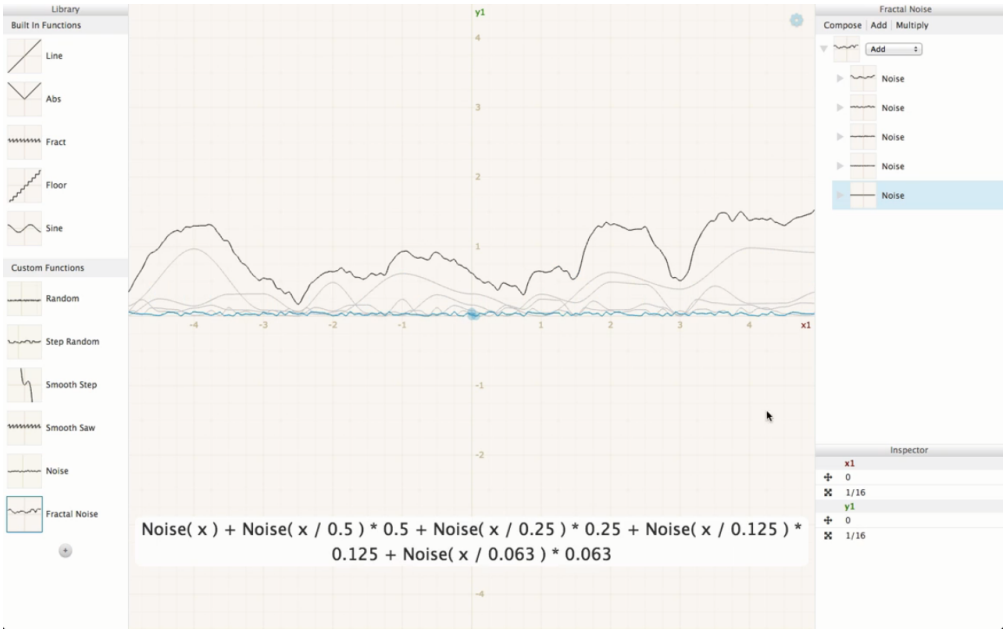


Figure 2.3: Above is the Shadershop interface, here used to create noise. This is done by taking all the functions on the left hand panel and combining them to create a low frequency noise. That is then added to subsequent harmonic series of the same noise in order to produce the pattern above.

Shadershop is a tool developed by Toby Schachman for the Communications Design Group made to facilitate function composition. The idea is to take simple functions such as a line or absolute value and compose

[1] The original sparse convolution process, without the convolution of the Gabor kernel was originally proposed in the late 1980s by Lewis. [Lew89]

them in different combinations in order to achieve a desired result through function composition.

For example, in order to produce a parabola (x^2), one would drag a line onto the editor twice and define their relationship as "multiply". Both lines would be seen in the background, and are editable - meaning one can rotate one of the lines and the parabola would then be affected by that change [Sch15b].

The elegance of this design is in its simplicity. It attempts to communicate the complex process of function composition in simple terms, by taking small well understood functions and abstracting upon them. This allows the user to understand the functions they are meant to compose from the ground up, rather than attempting to tease out a complex formula. The user is fully able to build their own personal abstractions in order to create their desired effects allowing for immense control without losing the ability to understand its component parts [2]

Yet some of its abstractions fall short of making a direct connection to the final result. The construction of the parabola for example, brings up several questions. The first is, to what extent is the notion of a parabola connected to the notion of two lines? Though the mathematical representations of $y = x$ and $y = x^2$ are similar, and one can easily be understood by the other, the visual representations of the functions are *qualitatively* different. Visually, a parabola might be described as a bent line - and it is not directly apparent how two lines can come together to create a parabolic function. Thus the use of two lines to represent a parabola could confuse rather than elucidate the notion of a parabola. With addition this abstraction might hold visually, as one can see how two lines are combined, but as with the parabola multiplication could not easily understood on this platform.

Nevertheless, the interfaces concentration on having the user construct functions from scratch is an important lesson from this interface - and one which my framework will concentrate on heavily.

2.4 ShaderForge

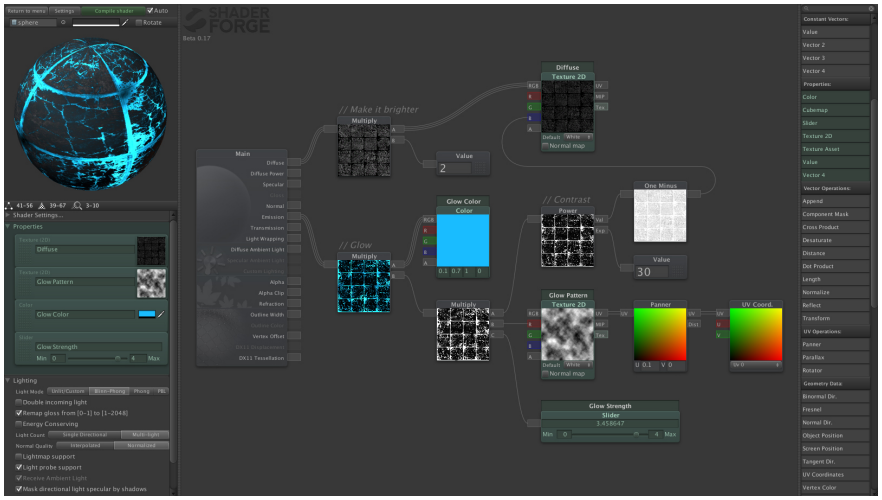


Figure 2.4: Caption

[2]Schachman presents an interesting counterargument while working with a slightly different interface, one meant to design recursive structures. He explains *geometrically* how adding two parabolas results in another parabola rather than any other shape, by talking through how his interface can be understood. While his explanation somewhat defeats the purpose of having a clear interface, it is nevertheless an interesting approach. He shows that by manipulating the functions and observing each at discrete points one can both see and understand the effect of each function at each given point. [Sch15a]

Shaderforge is an add on for the Unity ©engine, and is an example of a typical tool used to make textures. In essence, the interface functions as visual programming language where each node has inputs and outputs. The nodes are then chained together to create a pipeline, which subsequently produce a shader. This is similar to many tools included in 3D animation programs, where the main idea is to simplify the creation of shaders [Hol15].

Each node in the interface can be edited via a dialog box on the left hand side then put into the interface and chained in such a way so that it can be mapped to a certain type of light. The result is then shown in the upper left hand side of the screen, reflecting the live changes made in the interface. Each node has a preview meant to represent what kind of function, value, operation, or constant is stored in the node.

The main issue with this interface is its limited ability to be abstracted. As each node looks roughly the same, save the small preview window, it is difficult to tell the different operations apart. Furthermore, it is not easy to tell where each function begins and ends as the interface allows external inputs to littered throughout the design. The interface is also made rigid by not allowing users to make their own structures. It forces them to use largely hard coded nodes by connecting them to hard coded light values. Thus there is a considerable difference between the total node structure, the code, and ultimately the texture on the upper left hand of the screen.

That said, this type of interface for noise construction is very common as it is an easier way to convert an idea to a shader. The idea of using nodes is a very useful idea, and chaining nodes together gives a good understanding of how functional composition works. The framework and ultimately the interface proposed by this paper will be a design centered around this kind of node based interface - but with a concentration on small easily abstractable parts (as seen in shadershop).

Chapter 3

Perlin Noise

(a) Perlin Noise

(b) Fractal Perlin Noise

Figure 3.1: An example of how Perlin Noise can be mapped directly to a texture (3.1a), or mapped onto a texture after several fractal sums have been applied (3.1b) [OHH+02]

Originally introduced in Ken Perlin's 1985 paper *An Image Synthesizer*, Perlin Noise quickly became ubiquitous in the field of computer graphics, owing to its intuitive algorithm and wide array of applications.

3.1 Noise

Noise is a pseudo-random number generator, primarily used in computer graphics when extensive detail is required without necessitating a concretely defined structure. It allows for the introduction of randomness without sacrificing either continuity or control over spacial frequency [PH89]. This means that noise can be random and also have some control over the frequency distribution of these random numbers for a given distance, area, or higher order dimensional space.

In essence, it is a mapping from R^n, an n-dimensional point with real coordinates, to R, a real value [OHH+02].

More succinctly:

> A noise is a stationary and normal random process. Control of the power spectrum is provided, either directly, or through the summation of a number of independent scaled instances of (typically band-limited) noise [LLC+10].

A *stationary* random process is one whose statistical character is invariant to translation. In fact Perlin's Simplex Noise goes further and is also *isotropic*, meaning its statistical character is invariant to rotation as

well [Per85]. A *normal* random process is one where the probability density for any value within the set approaches a normal distribution across a band-limited spectrum. The summations are a clever trick used in many different implementations of noise in order to produce finer and more interesting details, best described as controlling noise in the frequency domain. This is usually referred to as *fractal noise* (see figure 3.1b), as these summations mimic fractal patterns [Gus12].

Noise is typically used to provide detail to textures, objects, and even animations. In many respects noise attempts to do what cannot be easily done by hand, and is an incredibly useful tool for attempting to emulate *stochastic* (randomly determined) natural formations such as marble, tree bark, or fire.

Types of Noise

Modern implementations of noise functions typically use one of three paradigms:

- Procedural Generation
- Explicit Generation
- Sparse Convolution Generation

Procedural noise generation is characterized by a technique where *algorithms* compute gradient values, the resultant values of a noise function, rather than copy gradients from a data structure. In other words, the gradient values do not explicitly come from a concrete source like picture or texture — but instead are generated "on the fly". Perlin Noise is an example of a type of procedural noise algorithm.

Conversely, explicit noise pre-processes and stores gradient values. The pre-processing of the noise itself can be procedural, but the noise function is not computed at the same time as it is accessed. Rather, the noise is stored in a data structure and referenced during the execution of the main program. This is the equivalent of using an image as a noise texture, and referencing it when creating geometry. Many techniques for creating noise rely on an explicit result, such as wavelet and anisotropic noise [LLC+10].

Lastly, sparse convolution noises generate noise by summing randomly positioned, oriented, and weighted kernels. The kernels are then convolved together in order to create highly controllable noise in the frequency domain. Gabor Noise is a good example of this approach. Originally created to improve the spectral properties of other noise generation methods, namely the ability to be anisotropically filtered, it is based on space convolution using the Gabor kernel — a kernel which can be easily parameterized as shown in Lagae et al.'s specular-based noise interface [LLDD09].

Procedural Noise

Along with the definition provided above, Lagae et al. lists several qualities of procedural noise[1]:

- The noise algorithm should use very little memory. The noise should theoretically take little to no space in memory, as the gradient values are not stored in a data structure.
- The noise should be "visible" at *any* resolution, meaning one can zoom in or out indefinitely without losing any of these properties.
- The noise should cover the entire dimensional space yet remain non-periodic, meaning without unwanted repetition or seams.
- The noise should be easily parameterized, allowing the power spectrum to be controlled thereby allowing the noise to be used for varying purposes.

[1]These characteristics are largely implementation dependent. Put another way, the following is a list of *potential* advantages

- The noise should be arbitrarily accessible, meaning any R^n coordinates should be able to access its corresponding R value regardless of where the value is in space, and in what order it is accessed.

[LLC+10]

Together these properties define a type of noise that is best used for dynamic content. Representations of terrain could zoom in our out indefinitely, never losing detail. One could translate or rotate the terrain and preserve the quality, while at the same time having control over spectral properties of the terrain via parameterization — and all without any pre-processing.

3.2 Classical Perlin Noise

Perlin Noise is one of the most common examples of a procedural noise function. As a result, it has been the subject of extensive study and review since its original publication in 1985. Many of the techniques Perlin initially introduced in his paper were later found to be inefficient, but the essence of his algorithm remained largely the same. Thus, in order to understand the modern use of Perlin Noise, his original technique must be considered.

History

Perlin began experimenting with noise after working at the Mathematical Applications Group Incorporated (MAGI) on Tron, a 1982 Walt Disney film about a programmer who became transported into the digital world of a video game. Tron was the first film with a large amount of solid shaded computer graphics, meaning many scenes in the movie contained mostly digital geometric objects which were then rendered with actors.

The aesthetic for the film, while interesting, was designed around the known limitations of the technology. The film did not use polygons, as many graphics engines do today. Rather, Tron was built from boolean combinations of mathematical primitives - ellipsoids, cylinders, and truncated cones. By adding and subtracting these shaped together, MAGI was able to create an interesting looking world with objects such as the geometric "light bikes."

At the time memory was a huge limiting factor, so large textures could not easily be used. Thus, Perlin began to experiment with procedurally generated noise functions to give the same "controllability" as the geometric shapes, but also introducing a level of randomness [Per99].

The Algorithm

The algorithm works by mapping textures directly to objects. It situates them within a field, and uses that field to manipulate various attributes. One way of thinking about this is by imagining a texture of the same dimensionality as an object, and mapping that texture to the object. Thus spacial coordinates can be used to index this new N-dimensional texture — freeing an object from the translation involved in, for example, UV-mapping.

Integer Lattice

One begins with the set of all points in space whose x and y coordinates are integer valued. This is a grid or *integer lattice* (Fig. 3.2). This amounts to splitting the N-dimensional space into an NxN grid of hypercubes (squares, cubes, 4D-Hypercubes, etc.). Note that this allows the algorithm to be abstracted to N-dimensions.

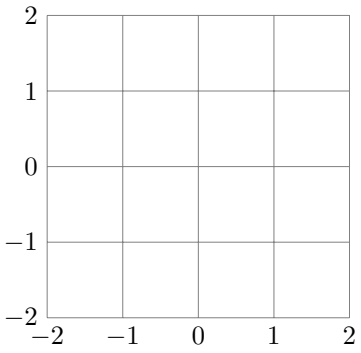


Figure 3.2: A slice of the integer lattice

Assign Gradients

Each point on the lattice is then assigned a pseudo-random gradient centered on each lattice point (Fig. 3.3). The gradient contains N scalars, one for each dimension. In three dimensions, this means each corner will be given three pseudo-random values between 0 and 1.

The representation below is a 2D visualization of the integer lattice where each corner of the lattice field $C(x, y)$ contains a vector which begins at C and ends at $(C(x) + gradient(0), C(y) + gradient(1))$.

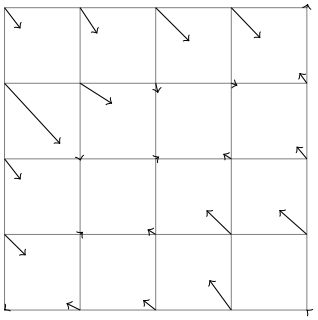


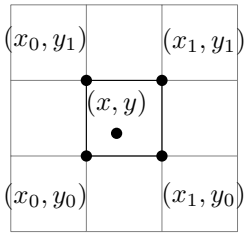
Figure 3.3: Integer lattice with pseudo-random gradient vectors

Find Gridpoints

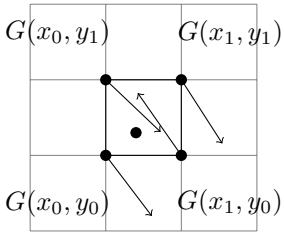
Find the current lattice square. This is done by taking the floor of the x and y values to find the low x and y bounds, and adding one to get the high x and y bounds (Fig. 3.4a). Then use those values to index the gradient function and find the gradient for each individual corner (Fig. 3.4b) [Per85].

Indexing the Gradient

In Perlin's original algorithm, indexing the gradient uses a hash value. His method requires two precomputed arrays: P and G. P contains a pseudo-random permutation of 256 (i.e. 0–256 in pseudo-random order), and G contains 256 pseudo-random unit length gradient vectors. A 2D gradient value for (x, y) is given by $G[P[(P[y] + x)\%256]\%256]$. The idea is to use the coordinates to repeatedly index the pseudo-random permutation table, thereby giving the algorithm its stochastic quality.



(a) Illustration of a the lattice slice around a point, where: $x_1 = x_0 + 1$ and $y_1 = y_0 + 1$.



(b) The corner points are then used to index a function G, whose input is each corner point and the output is a gradient vector.

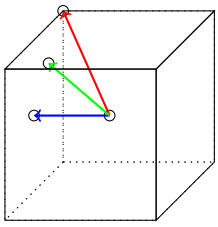
Figure 3.4

This model presented a problem, because gradients are chosen on pseudo random values between 0 and 1. This essentially constrains the distances that each gradient can be from the origin, "boxing in" the possible gradients. In 2D the difference is negligible, but when working in 3D the diagonals become considerably longer than a vector aligned close to the axes (see Fig. 3.5a). This results in a "clumping" effect, where values near the axes are much more similar than those near the corners of the cube.

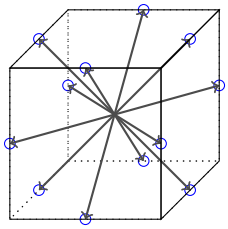
Perlin's optimization was to limit the amount of possible gradient values, so each vector only reached the midpoint of each edge of the cube (Fig 3.5b) thereby avoiding corners. The gradient vector array was thus shrunk from 256 to 12 values in 3D and could be in any order with sufficient randomness given by accessing the P table. What's more, lowering the amount of gradient values not only lowered amount of values to be indexed, but also the amount of computations. The indexes would only be 0,1, or -1, which allows these values to be easily multiplied [Per02]. At the time, multiplication caused significant overhead when processing an algorithm.

The resultant formula in 2D is $G[P[(P[y] + x)\%256]\%8]$ [Gus05], and in 3D is $G[P[(P[(P[y] + x)\%256] + z)\%256]\%12]$ [Per02].

.



(a) A vector that is aligned on the axis (blue) and moves toward the corners of the lattice (green) becomes significantly longer until it reaches the corner (red). The vectors would be the same length had they been extended to the unit length sphere surrounding the cube, but are cut by the lattice.



(b) Perlin's solution is to use a finite number of gradient vectors that begin at the center of the cube (the lattice corner) and extend to the center of each line on the sphere thereby resulting in vectors all roughly the same length [Gus05].

Figure 3.5

Scalar Product

Once found, the scalar product is calculated between each of the four gradient vectors $G(x_n, y_n)$ and the vector from the corner of each lattice point to the original (x, y) point, called the *Gradient Ramp* [Gus05]

(Fig. 3.6). In 2D, this results in four values that have both a relation to the random gradient, and the (x, y) point.

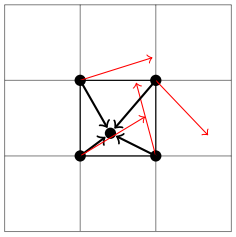
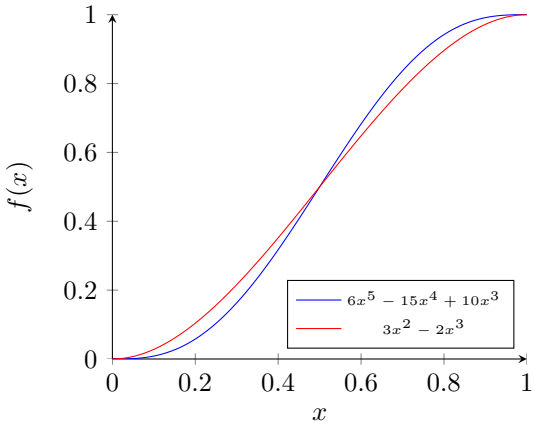


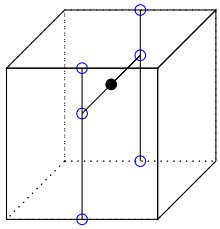
Figure 3.6: The dot product is taken between the gradient vectors (red) and the gradient ramp (black), resulting in four scalar values.

Interpolation

The resulting four values are then interpolated. This is done first in the x direction, and the result is then interpolated in the y direction. In Perlin's original paper, he proposed a third order S-shape interpolation (the Hermite blending function) using the formula $3t^2 - 2t^3$ [Per85], as linear interpolation would not provide a smooth enough transition to maintain continuity. He later found that the cubic interpolant function's second derivative $6 - 12t$ was not zero at $t = 0$ and $t = 1$. This introduced some discontinuity between lattice fields. He later revised that formula and proposed $6t^5 - 15t^4 + 10t^3$ as the interpolant function which has zero as its first and second derivative when $t = 0$ and $t = 1$ [Per02] (Fig 3.7a). In three dimensions this works similarly, seven interpolations would have to take place (Fig. 3.7b).



(a) Above are the two interpolation functions, Perlin's original third degree polynomial (red), and the revised fifth degree polynomial (blue). Notice that the fifth degree polynomial provides a smoother curve.



(b) Above is a representation of 3D interpolation. The gradient for the (x, y) point, shown in black, is found by interpolating across the corners of the lattice, then interpolating across the resultant values. This results in a total of 7 interpolations [Per99], with 24 multiplies [OHH+02].

Figure 3.7: Interpolation

The result of the interpolations is the final noise value used for the given (x, y) point [Gus05].

3.2.1 Problems with Classical Perlin Noise

While the algorithm had been successful in creating reliable procedural noise, Classical Perlin Noise has one significant deficency:

Difficulty with generalizing to higher dimensions In two dimensions, a single lattice area is a square with four corners. In three, a lattice area is a cube with eight corners. In four dimensions, a lattice area is a 4D hypercube with 16 corners. This can be generalized to say that a hypercube in N dimensions has 2^N corners [Gus05]. Thus Classical Perlin Noise is an $O(2^N)$ problem, meaning each successive dimension is at least twice as inefficient as the last [OHH+02].

3.3 Simplex Noise

In 2002 Perlin improved the performance of his original noise algorithm by creating his patented [Per05] *Simplex Noise*. Simplex Noise does not produce exactly the same results as classical Perlin Noise, but Perlin claims it provides the "same general 'look' as previous versions of noise", while working much more efficiently [OHH+02].

In essence, Simplex Noise reduces the amount of computations by changing the algorithm's understanding of the lattice field. This results in fewer corners and fewer calculations. What's more, because of the special relationship between vertices in Simplex Noise, the resulting corners are summed and not multiplied — again a very important efficiency consideration when the algorithm was conceived.

Simplex Grids

(a) A lattice field

(b) A Simplex Grid

Figure 3.8: The light and dark gray areas on the Simplex Grid (Fig. 3.8b) are a skewed version of the same shaded areas on the lattice field (Fig. 3.8a) [Gus05] where for every point in the light area $y > x$ and for every point in the dark area $x > y$. The red lines in Fig. 3.8b subdivide each square until the space is composed of Simplex shapes, in this case, equilateral triangles.

The crux of Simplex Noise is the *Simplex Grid*, a simplical tessellation of N-Space. A Simplex Grid, like the lattice field, is also a method used to encompasses an N-dimensional space. The difference lies in how the space is tessellated.

For a given N-dimensional space, a Simplex grid is made up of the most compact N-dimensional shape which can be tessellated to fill the entire space. For a 2D grid, the simplest shape is an equilateral triangle (Fig. 3.8b). Likewise the Simplex shape for a 3D grid is a skewed tetrahedron. In general, for a given N-dimensional space its simplex shape has $N + 1$ corners, and $N!$ of these shapes can fill an N-dimensional

hypercube (squares, cubes, etc.). A Lattice field cell requires 2^N corners, because it fills the space with N-dimensional hypercubes , far more than its Simplex counterpart [Gus05].

One way of understanding this is by taking an N-dimensional lattice field and skewing it along its main diagonal. The skewed hypercubes can then be subdivided until the field is made up of only simplex shapes, as in Fig. 3.8.

The first step of the Simplex Noise algorithm is skewing the lattice grid (along with the N-dimensional point) along its main diagonal in order to produce an analagous grid in Simplex space.

Selecting a Simplex

Once the Simplex grid has been established, the Simplex containing the N-dimensional point must be found. The hypercube for a given N-dimensional point P can be found by taking the floor of each scalar value within P to find the lower bounds, and adding one to each value to find the upper bounds. The grid can then be converted back to the original unskewed coordinate system.

The indices are then used to find the Simplex. Depending on the magnitudes of each scalar within P, a Simplex location can be found. For example, for $P(x, y)$ the upper Simplex would be selected if $y > x$ and the lower Simplex would be selected if $x > y$ (see Fig. 3.8). This can be extended to N-dimensions by comparing the different values of each element within P.

Summation

Like in Perlin Noise, each corner of the Simplex containing P has a unit length random gradient vector $G_v c = G(P)$ where G is the pseudo-random gradient function (Fig. 3.4b) and c is one of the $N + 1$ corners of the Simplex. The scalar product is found between the gradient vector $G_v c$ and the gradient ramp $G_r c$ (Fig. 3.6) to obtain a scalar value S_c. In Classical Perlin Noise, each S_c would be interpolated to find the resultant scalar value at P, but Simplex Noise takes a different approach.

With Simplex Noise, each S_c value is instead multiplied by a radially symmetric attenuation function to get a weighted scalar value. In essence, this means that each S_c will be multiplied by a weight w_c that is representative of each corner's distance to P. In an implementation made by Gustavson [Gus05], he calculated w_c in 2D as:

$$w_c = (max(\frac{1}{2} - (Dx_c^2 - Dy_c^2), 0))^4 \tag{3.1}$$

Where Dx_c and Dy_c are the distances in the x and y directions between each c and P. Only the c corners affect a point found within a given Simplex as any other vertices in the Simplex grid will decay to zero before reaching the current Simplex. Each corner of the Simplex is at an equal distance, and as the values are already weighted based on their relation to P, there is no need to interpolate (Fig. 3.9).

The result is the summation of weighted values for each $N + 1$ corners of the current Simplex:

$$\sum_{c=0}^{N+1} w_c * s_c \tag{3.2}$$

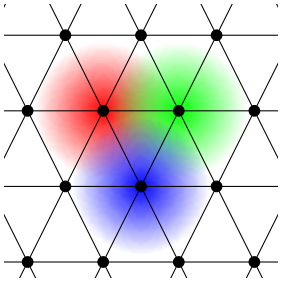


Figure 3.9: Each point within a simplex is influenced by a symmetric radial attenuation function. The symmetric radial attenuation of each corner of the center simplex is visualized here as a color. The closer a point within the simplex is to a corner, the more *weight* that color has on the point. Each color is multiplied by that weight, and the weighted colors are summed to give each point's individual gradient.

3.4 Using Noise

.

Noise is typically used to introduce a controlled pseudo-random element onto some environment. Shaders are best suited for this task because they are built directly into the Graphics Pipeline, allowing each point's color and position to be calculated procedurally. Procedural noise is arbitrarily accessible (see page 9), making it is a perfect candidate for modern Shaders[2], and is often implemented with the OpenGL Shading language (GLSL)[Gro15][3].

Below are some common examples of how noise functions are used. The noise function, which takes an N-dimensional vector R^n and returns a scalar value R, hereafter is referred to as $Noise_n()$.

Solid Texture

Traditional approaches to creating noise with Computer Graphics relied on mapping color to textures and subsequently mapping those textures onto an object. Perlin Noise on the other hand uses the concept of a *Solid Texture* to build objects. Objects built with a Simplex or Lattice grid are not objects whose noise attributes are mapped, but rather are isosurfaces of a solid noise field.

Thus an N-dimensional object has been *sculpted* by the noise of the grid around it. The key advantage is that a texture does not need to fit directly onto the object, objects can change shape and be carved into pieces at will — while retaining the same noise qualities of the original object [Per85].

3.4.1 Noise Composition

Mapping Noise

The most straightforward application of $Noise_n()$ is to map the noise directly to a texture. One can do so by coloring each pixel with a noise value band limited by two intensities:

[2]Perlin even developed specialized hardware specifically geared toward calculating noise. The emphasis is on an efficient pipelined parallel implementation [OHH+02]

[3]Note that *built in* noise has been depreciated with version 4.4 of GLSL

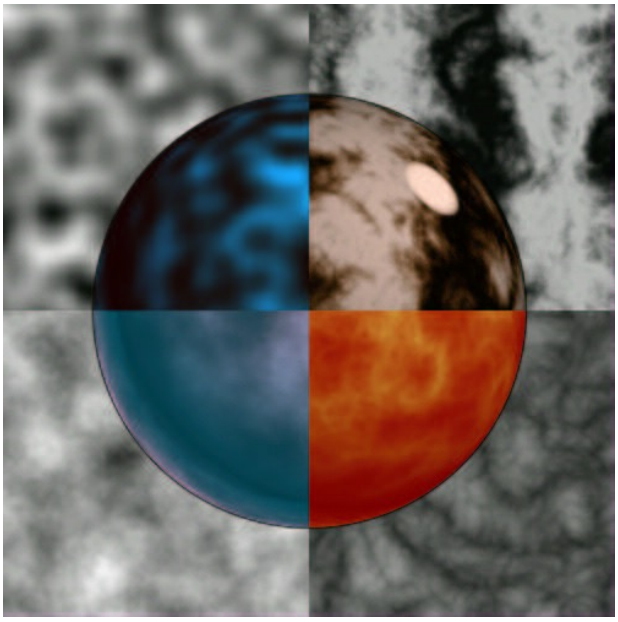


Figure 3.10: Each quarter of the image represents a different technique for creating noise [OHH+02]. Clockwise from the top left is a visual representation of Listings, 1, 4, 3, and 2. Note that the color vectors and lighting models used in the image differ than those found in Listings — but the underlying algorithm is the same.

```
# noise() is equivalent to Noise_3(), whose band limits are 0 and 1;
vec4 whiteColor = vec4(1.0, 1.0, 1.0, 1.0);
gl_FragColor = whiteColor * noise(position);
```

Listing 1: Pseudo-random noise mapped to a texture. Note: color is mapped using RGBW — with W as the Homogenous coordinate. Visual representations of noise Listings shown in Fig. 3.10

Fractal Noise

One can sum octaves of noise to produce a simulation of a $1/f$ signal (fractal noise):

```
# n_sum() is equivalent to \sum_t Noise(point * 2^t) / 2^t
gl_FragColor = whiteColor * n_sum(position);
```

Listing 2: Fractal noise. Note that gradients of $1/f$ functions are similar at all scales [Per85], which means the amount of octaves to be added depends on the scale each pixel is on the texture. In theory, one can then zoom in or out indefinitely without a change in statistical properties of the noise.

Turbulent Noise

Turbulence can be added to the noise, in order to distort it further. Applying the absolute value to each octave of fractal noise ensures that the noise will have discontinuous *boundaries* at all scales, while keeping its continuity. This results in visible "wisps", or what Perlin calls "discontinuous flow" [Per85]. Below is an example implementation:

```
# Turbulence function
float turbulence(vec3 point) {
  float t     = 0.0;
  float scale = 1.0;
  while (scale > pixelSize){
    t += abs( noise( point / scale ) * scale);
    scale /= 2;
  }
  return t;
}

# Within the main function
gl_FragColor = whiteColor * turbulence(position));
```

Listing 3: Turbulent Noise

Striped Noise

Regular procedural patterns can be used in conjunction with Perlin Noise to create more interesting effects [Gus12]. A regular procedural pattern could be something like lines or grids, composed with repeating functions such as a sine wave.

Below is an example of how a regular pattern can be used with turbulant noise in order to get a "marble-like" effect. The regular sine function is composed with the turbulence function to create noise superimposed on stripes. This kind of function composition is key to designing interesting noise functions:

```
# Creates horizontal stripes
gl_FragColor = whiteColor * sin(position.x + turbulence(position)));
```

Listing 4: Striped Turbulant Noise

More examples

For more examples of the use of Perlin Noise, please refer to the appendix as it contains several examples of how functional composition can create different types of textures (Figures A.1 through A.5).

Chapter 4

Framework

The noise interface design will be geared toward creating 3D textures using function composition. The goal is to be able to have users understand how to work with noise by augmenting other functions with noise. As such, the interface must allow for each function within the composition to be so well understood that it can be used even with the addition of something as unpredictable as Perlin Noise.

This is the challenge of designing for this type of interface, as no part of the resulting structures should be abstracted to a point where it is no longer understood ("black boxed").

Computers have attempted to abstract complicated processes since their inception, and many techniques have been developed to bridge the gap between some of these complex ideas and those of the user. That said, there is not a concrete set of design principles specific to creating abstraction-oriented interfaces. Thus in order to overcome the aforementioned problems with creating an interface for noise, the interface must have its own design philosophy - based upon some of the principles of other systems and system designers.

Design Philosophy

The main philosophy guiding this framework will revolve around two critical ideas:

- Composable Design
- Semantic Abstraction

Together these ideas help define a design framework which balances usability and complexity in such a way that allows users to be truly creative with noise.

4.1 Composable Parts

The Unix Approach

A good example of a system which sucessfully implements a composable design is the Unix system. Developed at the Bell Telephone Laboratories, it has since become one of the most common operating systems in the world. While it is an operating system - many have also come to think of it as a design standard. This is because the command interpreter's syntax, language, and organization communicate and allow for much more than simple commands. The crux of its design is that it is constructed out of small concepts and programming modules. These modules are then interconnected via pipes[1] to make new functions. Those

[1] A structure that directs the output of one input to the input of another.

functions can then be piped to other functions, which allows the user to define very specific tasks with very few commands.

4.1.1 Unix Design Standards

The composable nature of the Unix system is made possible by having two key design principles: uniformity and small generalized parts. Uniform design is the notion that no part of the system is defined in so specific a fashion that it cannot be interconnected with another part of the system. For example, in Unix there is no difference between an I/O channel, a file, or any other system object. This in turn allows these objects to be connected in seemingly any way, without having to worry about what type of object is being edited. The composed commands can then be put in a file and referred to as a separate command, which again can be compsed and used again.

The glue that connects these pieces together is the small Unix programs. These programs are very small and specifically defined, meaning they typically have a single lower level functionality which outputs either nothing or exactly what it was meant to (no extraneous outputs). They allow for Unix programs to be very flexible, and facilitate their ability to be interconnected at will without sacrificing key functionality. Yet, in its attempt to create the most concise core from which to build functions, it has left some users behind. By not allowing users to use abstractions they had not themselves built, the system ties the user to the sometimes overly specific functions.

4.1.2 Critique

For example, in a 1981 issue of Datamation Magazine, Donald A. Norman[2] described the Unix system of the time with some contempt.

He began by claiming that Unix's naming schemes were inconsistent. In Unix, commands like move (mv) and link (ln) are abbreviated while others like echo and date are not. He also belived the names of functions did not always have to do with the functionality - like the dsw command. Dsw is a command that lists all files and asks if they would like to be deleted. The true meaning of dsw, though not present in the manual, was later revealed in the article as "delete from switches" - again having no real tie to the functionality of the system.

Yet his most salient point, and one that gets to the heart of this type of design, was his criticism of the "friendliness" of the Unix program. Unix programs are designed to be piped, not to be read. Thus if one searches for a file or asks to list the contents of a directory, or even appends information to another file - there is no information given to the user that some successful operation has taken place. The action either occurs or does not.

In the same paper, Michael Lesk at Bell Labs offered an interesting point. If a command lists the contents of a file with a heading, the next command would have to strip the heading for its own output - and if that command then adds its own header - the problem quickly snowballs [Nor89]. Thus the question remains: How does one design a system that is at once understandable and neatly composable? This issue is inherent with this kind of design, and is what this framework seeks to improve upon.

4.1.3 Composable Design

[2] A psychologist and designer, probably best known for publishing the book "The Design of Everyday Things"

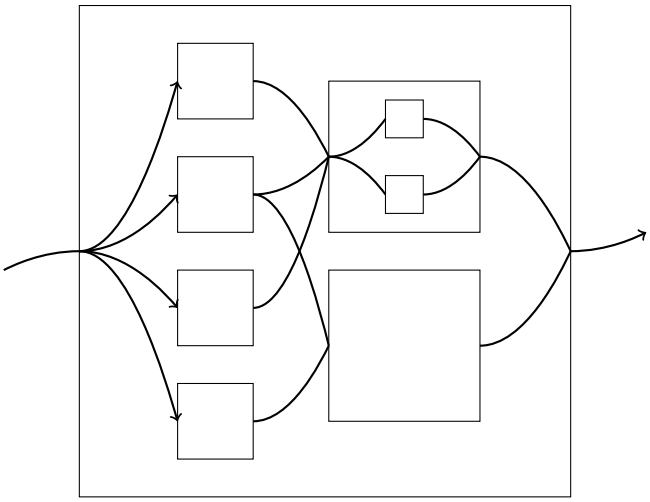


Figure 4.1: In the image above, each square is node (function) which connects to another node. The nodes take any number of inputs and outputs, and can be rearranged at will. The outside node is no different from the internal nodes. Like a Unix shell script, it too is a node which in this case with one input and one output.

While Norman's specific points can be debated,[3] his larger concerns about readability and transparency within a composable system is an apt critique. The noise interface will be an attempt to foster the trust and comfort of the user by both offering the composable design which made Unix such a powerful system - but adding exactly what Norman was looking for by giving users immediate, understandable feedback without sacrificing the ability for functions to be chained together in an elegant fashion.

The noise interface will attempt this Unix-like approach. Small sections of generalized shader code will be chained together via connections (pipes). These pipes will then be connected to other sections, which could be the original sections or a set of sections connected via pipes. Visually, this is best seen as "zooming out" from a function - starting from a small specific function to many complex functions chained together in varying ways as shown in Fig. 4.1.

4.2 Abstraction

In order create a system where functions can be tied together in a seemingly infinite fashion, it requires the ability to abstract relatively easily. If the program grows faster than can be understood, it runs into some of the problems Norman was describing above - namely it loses it's "friendly" quality.

Thus the functions must be encapsulated in a manner which the user can most easily understand, and one which does not compromise the pipeline design or the integrity of the small generalized functions. The theoretical framework for how the interface is meant to abstract and communicate functional composition effectively will be based on the ideas of Seymour Papert.

4.2.1 Papert's Principle

Seymour Papert is a seminal figure in computer science education[4], and lends several ideas for how to design systems with users in mind. His approach is to delve into how users learn, and design systems based on how to engage users to teach themselves.

[3] For example, whether uniform abbreviation facilitates understanding has yet to have been proved.
[4] Among several other fields

Papert focuses on how users engage in a conversation with artifacts[5], and how those conversations can encourage self directed learning, lifting the burden from the system itself. Artifacts, he believes, should allow users to build their own notions on how best to use them, rather than outwardly teaching the user every single step.

He argues that if users are given only the ideas or explanations that a system provides, it can inhibit the user from being able to learn. They would always be attempting to *translate* another person's ideas rather than simply understanding the artifact in its own right. If on the other hand the user creates their own abstractions and structures, they can engage in the material in a wholly different way. Thus he argues for a shift in thinking, from a "universal" design to an "individualized" design[6] [Ack01].

Marvin Minsky described what he called "Papert's Principle" as the following:

> Some of the most crucial steps in mental growth are based not simply on acquiring new skills, but on acquiring new administrative ways to use what one already knows. [Min86]

The noise interface will take this lesson from Papert. It will use abstractions that can be both well understood and allow users to personalize content. Users should be able to organize each function (or node, using the example from Fig. 4.1) in a way that, as Minsky put it, allows them to use what they already know in a different way.

4.2.2 The Semantic Abstraction

In order for the interface to encourage the kind of dialouge Papert's principle requires, the structure of the nodes needs to be easy to understand and flexible. This is where the idea of a "semantic abstraction" is of use - it is an abstraction made to hide away details, whose meaning is tied directly to what it is attempting to abstract.

A good example is animation. If several menu items are sub-menu items, one can click on a menu item and see the sub-menu items emerge out of the parent. No other explanations are needed, as the user is using their *previous knowledge* about how one object can contain another, and using it in a *new context* to represent menu items.

Likewise, the interface will rely on this kind of semantic abstractions by using scale to represent granularity of the function. At the lowest level (smallest scale), one is able to directly edit the provided shader code. This allows users to either use the packaged shader code or edit it in order to create a more specific design. One can then zoom out until the code is visually obscured in order to see how the function connects to other functions within that scale. As the user continues to add nodes, the smaller components fade away - but do so in a way that is *semantically* tied to the concept the abstraction using. Thus, the user will be able to have Papert's dialouge with the interface because the nodes can be organized and abstracted upon at will, and in a way that uses what the user already knows about scale and granularity.

4.3 Design Principle

These two notions, that of providing a composable interface and that of creating semantic abstractions, require a careful balance. A fully composable interface would be akin to a Unix like code-based piping system. A fully abstracted system might simply be drag and drop textures from a menu onto an object. Each has its advantages - but a good set of design principles for working with procedural noise with function composition requires a middle ground. Put another way, one should be able to picture and edit the smallest detail and largest structure at the same time.

[5] An artifact can be a computer, an interface, or any other object the learner interacts with.
[6] Put another way - A design which can users can *relate* to individually.

This is the design principle of the noise interface: to encourage users to explore and understand on their own by making a system as transparent and compsable as possible.

Chapter 5

Prototype

The prototype is essentially a proof-of-concept of the aforementioned framework. While far from a complete interface, it provides a scaffold for how a designer can empower users on how to work with complicated concepts such as noise and function composition. By incorporating the use of a highly composable design, coupled with the implementation of strong visual metaphors (semantic abstractions), the prototype interface attempts to create an environment which affords the user with the maximum amount of creativity and freedom, without sacrificing the underlying concepts behind function composition with noise.[1]

5.1 The Node Graph

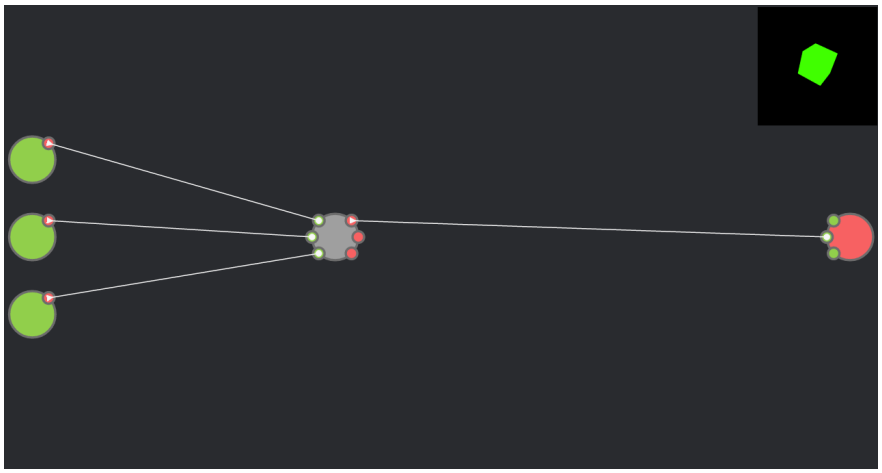


Figure 5.1: An image of node graph in its initial state, with three external inputs on the left and one external output on the right. The inputs (green) are the x, y, and z positions, while the output (red) is a color, represented in RGB. In the center is a single function, which takes the position and outputs three values, one of which is mapped to the second input of the output node, namely 'green'. A preview is shown in the upper right hand corner.

The prototype is a node graph where each node represents a section of GLSL code. These nodes are then

[1] A preliminary prototype was also created, one where nodes can only be used as color transfer functions. It proved to be much too rigid a design, and was replaced with the current iteration. More on the preliminary prototype can be seen in the Appendix (Fig. A.5).

chained together via their respective inputs and outputs in order to produce a fragment shader. The shader is then produced live and displayed in the upper right hand corner.

Inside each node, the code takes the inputs given to the node and produces a set of outputs. The scope of the code does not extend beyond the node, compelling the user to abstract their code and chain nodes together in order to produce a more complex result.

The isolation of the code sections allow a for a Unix-like approach, as the interface will provide most common code bits needed. Along with many basic algebraic and linear algebra operations, the interface could provide more abstract nodes such as a color transfer function and of course different types of noise functions. In theory, the user would never necessarily write any shader code, as the prepackaged nodes should suffice for most basic tasks.

In the prototype interface, these functions are specified with GLSL code — and many users will not need to edit much of the code. Like with Unix, the art the of the interface is in how these different functions connect, instead of how each individual node is constructed.

5.2 Code Blocks as Composable Parts

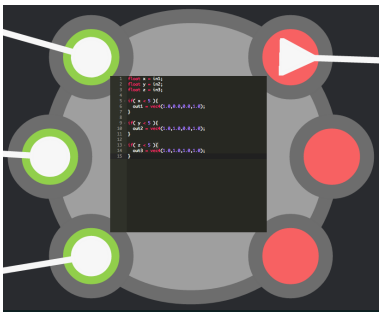


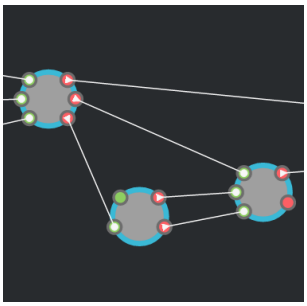
Figure 5.2: A single node when zoomed in reveals its underlying GLSL code.

Each node is a block of code, which is then composed together with other nodes to produce a procedural texture. The code blocks function similarly to individual Unix programs, where the user takes nodes and links (or pipes) them them from one node to another. It encourages the user to separate out parts of the GLSL code which may be too convoluted, and organize the nodes according to however the user sees fit. Yet, the interface differs from the Unix design.

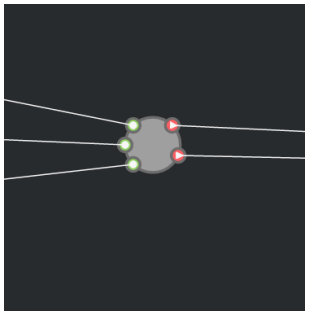
Each node is not necessarily as specific as a Unix program — as one can zoom into any node and edit the contents. This design was chosen because it was important to have the user be able to both use the packaged nodes (with pre-written code) *and* have the ability to construct or abstract on code they already knew. The user is not required to edit the code but by allowing the user to do so, the interface conforms to Papert's Principle. It allows the user to create foundations of their own design, and subsequently chain them together in order to create an abstraction personal to the specific user.[2]

5.3 Granularity as a Semantic Abstraction

[2]To use Minsky's language, the node would represent a concept and the graph would represent a *new administrative way* of using that concept.



(a) Three nodes are selected.



(b) They are then grouped, with all internal connections hidden from this scale.

Figure 5.3: An example of grouping.

An essential concept of the interface is that the node graph and the GLSL program are essentially the same. The difference lies in how the data is represented. The crux of the interface is its emphasis on structure, separating code out in order to limit the complexity of each code block. That said, users will require more complex structures in order to create textures such as marble, wood, or clouds (see Fig. 3.10). In order to create more complex graphs without adding more nodes or creating huge code blocks, the interface uses groups.

As seen in Fig. 5.3, once nodes are added to the graph they can be grouped together, abstracting all the nodes and links within a given group and connecting that group to any input and/or output with a source/destination outside of the group. The group then functions as any node, whose inputs and outputs can be rearranged at will. In order to evaluate or change a node within a group, the user would zoom in on a given node, exposing its underlying structure.

In this way, the interface uses granularity (or scale) as a way to encapsulate concepts together. For example, if one were to take a noise function and apply a harmonic series to it, one might want to group those nodes together in order to create a 'harmonic series group', thereby abstracting away the construction of the noise function.

Using scale to evaluate components is a way to enforce the semantic abstraction of grouping. The user understands both visually and semantically that inside any given group there are more nodes, and that inside any given node is the GLSL code. At no level does the abstraction of the group lose it's semantic connection to the way the underlying code is structured.

5.4 Noise Generation

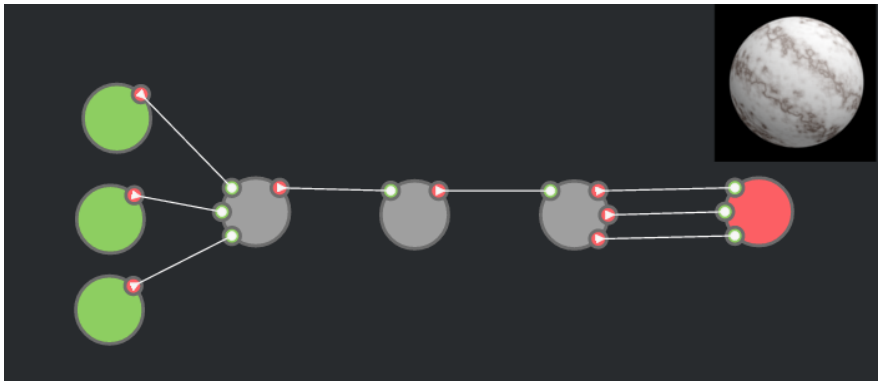


Figure 5.4: An image of how the node graph would be arranged in order to produce a marble texture. Coordinates are fed in from the green nodes to a noise node. The noise node feeds into a function which takes a float and augments the frequency of a sin wave on that float. Lastly, it is chained to a color transfer node, which outputs the RGB values used on the sphere.

Using this interface, noise generation becomes trivial. The interface would provide nodes for noise, such as Perlin Noise or turbulent noise, and those nodes would be chained together with other functions in order to produce the desired result. Composing textures using noise would be no different than composing textures using any other procedural method. A more concrete example can be seen on Fig. 5.4.

The interface is designed to have users thinking procedurally about textures, in that the output is always the chaining of functions from position to color. No noise-specific abstractions are required in order for users to work with and manipulate textures - it is simply one of many functions in the node graph.

Chapter 6

Discussion

The current interface is only a prototype, and far from the full potential this kind of an interface can achieve. That said, the initial goals of the design have been met. Users are encouraged to create their own designs for procedural textures. They do so by thinking procedurally and constructing sets of functions of their own design, based on code of their own design. They can then be easily chained and grouped to compose more complex functions.

The graph satisfies many of Unix's key design principles, and offer semantic abstractions which do not force the user to adopt notions they had not created themselves. Papert's Principle holds, in that the interface does not attempt to teach new concepts. Instead, users build each node graph from the ground up by using their own knowledge to create nodes, and connecting them in new and interesting ways to produce a procedural texture.

6.1 Critique

That said, the design is not without its flaws. Three key points, namely the transparency of the nodes, the lack of pre-built nodes, and the lack of a preview limit the potential of the interface.

6.1.1 Node Transparency

In the current design all nodes in each scale are the same shape, size, and with the notable exception of the inputs and outputs — the same color. This makes the nodes difficult to understand at a glance. In order to understand the difference between each function, a user would have to zoom in on each node, see if it is composed of other nodes, and analyze each bottom level GLSL component. This can be easily remedied by allowing the user to select different colors and labels for each node, specifying its functionality.

6.1.2 Pre-build Nodes

Currently, there is no place on the screen to accommodate the pre-built nodes. The interface requires a way to organize the pre-built nodes in a way that users can quickly access - and in a way that does not inhibit the users' ability to understand what each node's functionality is. A long list of node types for example, would be too verbose and unwieldy for the user - thus the user would require the functions to be organized by type and functionality, and easily searched.

6.1.3 Multiple Previews

While one preview at the end of the chain of nodes provides a good visual for the final output, the user is not able to see what the texture would look like at each step. For example, in Fig. 5.4, it would be useful for the user to be able to see a preview at each step of the process. If the user would like to change some aspect of the node which takes in the noise value and outputs the result of a sin function on that noise value, they would only be able to see the final result. Allowing the user to see multiple previews of the result at different stages gives the user a more fine tuned understanding of the node graph - and can help tie the final product to each step in the function composition.

6.2 Conclusion

By understanding the common ways noise is created and used in computer graphics, this paper sucessfully provided a framework which allows users to be creative with functional composition, ultimately designing with the use of noise. The stochastic character of noise required the interface to be understood and used intuitively, as an overly complex interface would simply not be an improvement over writing a shader program from scratch. Thus the interface succeeds by defining a way in which to use a Unix-like approach to shader composition, and by using scale as a semantic abstraction.

While the prototype interface can stand to be improved, the framework laid out in this paper allows users to begin having the dialogue Papert was referring to with procedural textures[1]. By allowing users to connect with the interface and relate to the node graph as something they had created, this dialogue can help users explore and discover new ways to work with procedural texturing with noise.

Thus, a framework has been produced which allows non programming-oriented users to be able to build and construct complex textures using noise and function composition. If the basic guidelines are followed, one can create any number of noise or function composition interfaces and have the material still be extensible, communicable, and adaptable for a larger percentage of users.

[1] See section 4.2.1

Appendix A

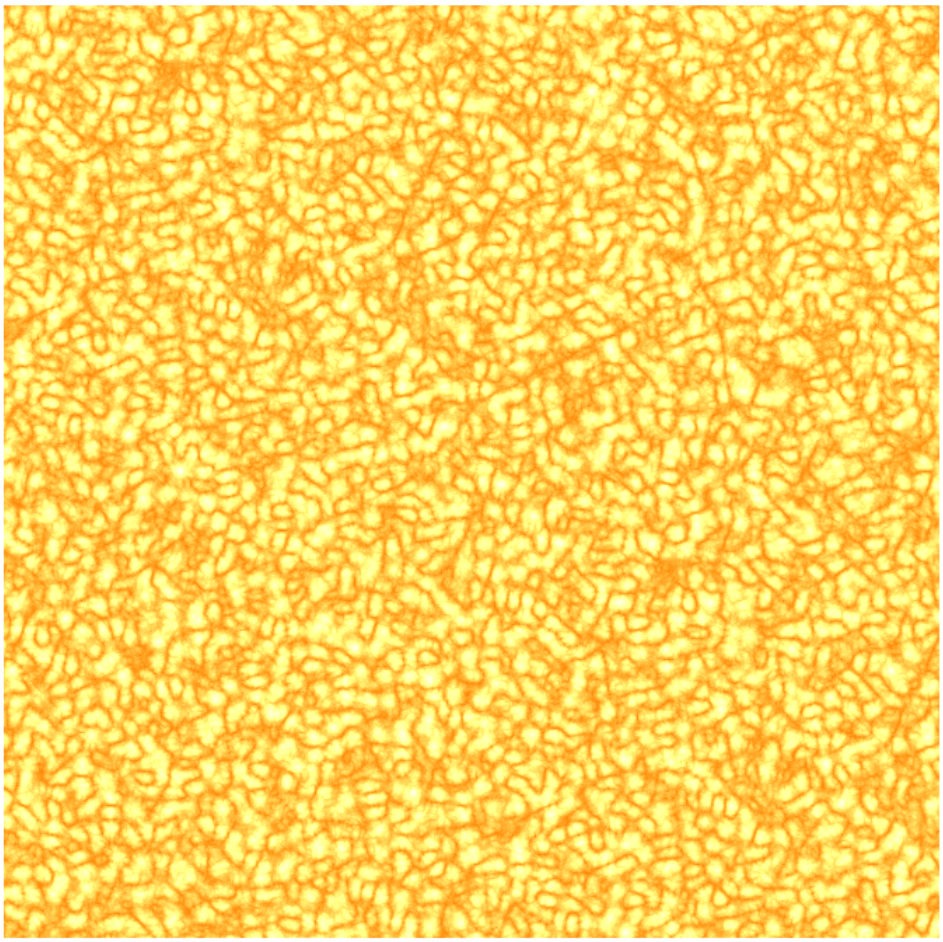


Figure A.1: An example of turbulent noise used as a color transfer function between white and orange. The result is then mapped to a plane.

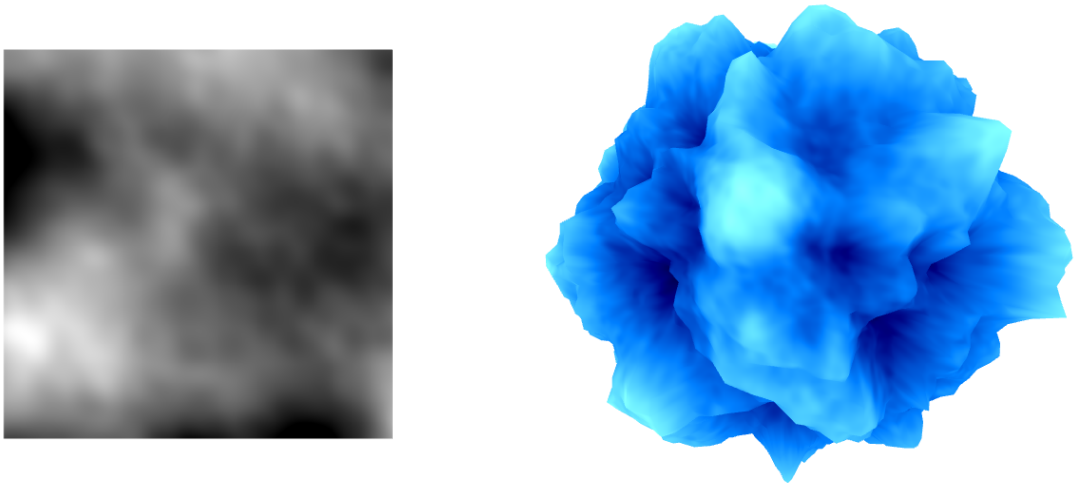
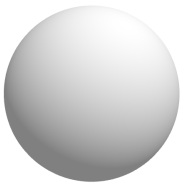
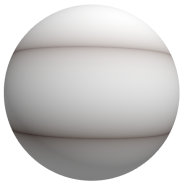


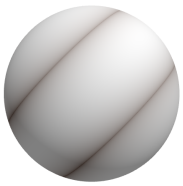
Figure A.2: An example of a harmonic series of noise mapped first to a plane, and then to the surface of a sphere. The plane can be understood as a "slice" of the sphere, showing that though the sphere is hollow - the texture itself is 3D. While the plane uses the resultant gradient values as a color transfer function between black and white, the sphere uses them to find displacement of each point along the normal, and interpolate between the colors black and blue.



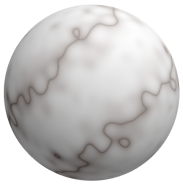
(a) A surface to be textured.



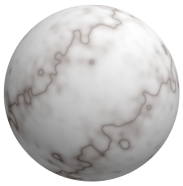
(b) The same surface mapped with a sine function (taken to a power) used as a color transfer function between white and brown.



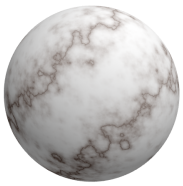
(c) The sine wave is then manipulated by rotating it along the z-axis.



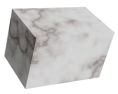
(d) An octave of turbulent noise is added.



(e) Another octave of turbulent noise is added.



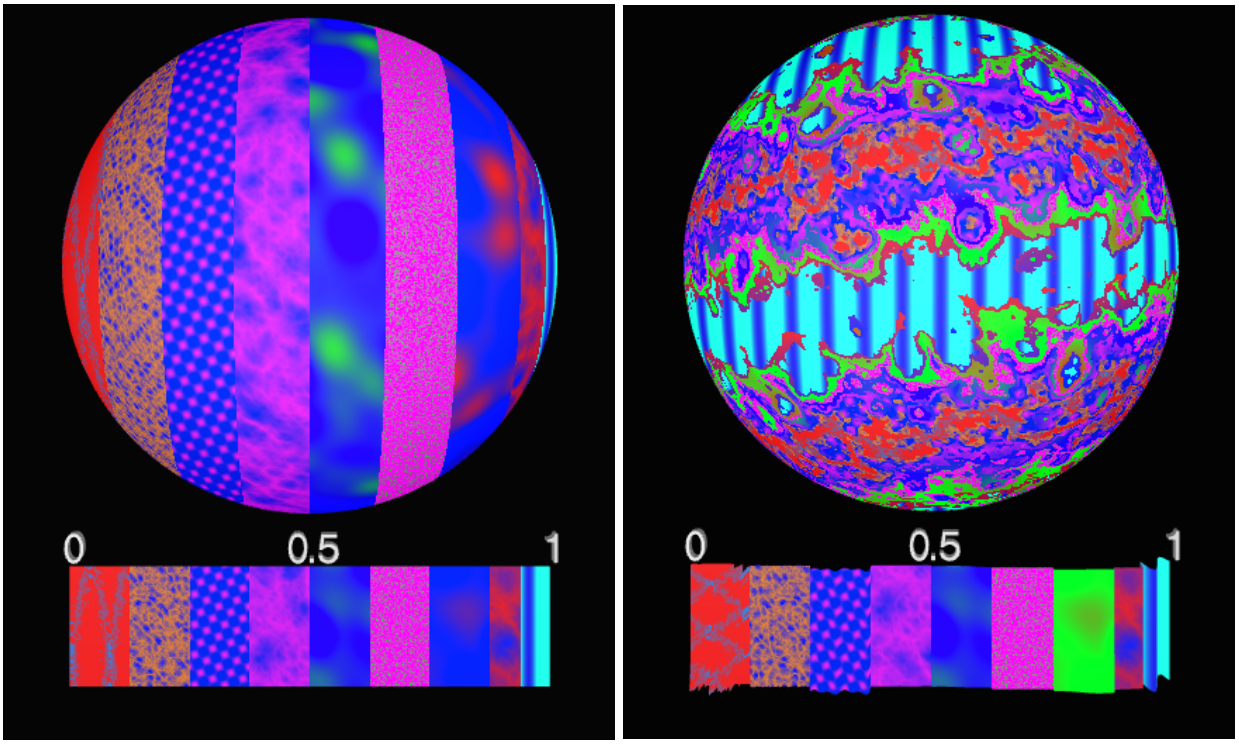
(f) One last octave is added, and the sine wave is taken to a slightly lower power.



(g) The texture can then be mapped to an object, like this rectangular cuboid. Notice how the texture retains its quality regardless of the shape it is mapped around.

(h) Another example, mapped to a cylinder.

Figure A.3: An example of how function composition can be used to create a marble like texture, and how it can then be applied to any number of objects while still retaining its quality.



(a) The color bar is mapped here to the normalized x-position.

(b) The color bar is mapped to a function similar to the one shown on Fig. A.3

Figure A.4: Two examples of how functional composition with noise can render complex and interesting results. In both examples a color bar is used which for each section interpolates between two colors using a composed function. Note that each function is composed only using some combination of a sine wave, a line, and some form of noise (Perlin Noise, harmonic series, or turbulent noise).

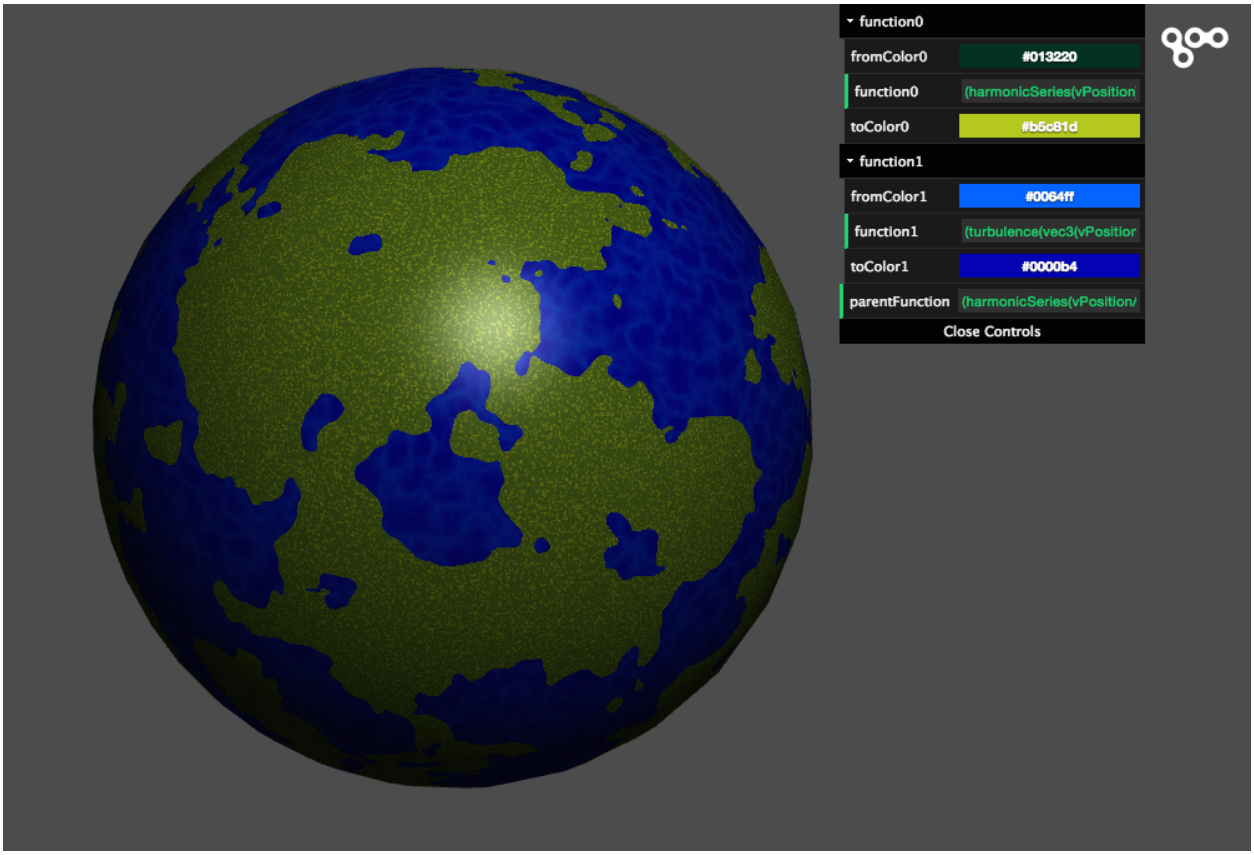


Figure A.5: The preliminary functional composer. Each "function" had three components: an interpolation function, and two colors to interpolate between. The user defined each color and manually typed out the GLSL code for the interpolation function. Finally, a parent function delineated which section corresponded to each function. The design was eventually scrapped because of its limited extensibility - one could only really abstract away the colors and functions, making it essentially an interface for the previous example (Fig. A.4). A node based system is highly preferable because it could be easily generalized to be a function, color, input, output, or anything else.

Bibliography

[Ack01] Edith Ackermann. Piagets constructivism, paperts constructionism: Whats the difference. *Future of learning group publication*, 5(3):438, 2001.

[BCA+14] Gwyneth A. Bradbury, Il Choi, Cristina Amati, Kenny Mitchell, and Tim Weyrich. Frequency-based controls for terrain editing. In *Proceedings of the 11th European Conference on Visual Media Production*, CVMP '14, pages 15:1–15:10, New York, NY, USA, 2014. ACM.

[Gro15] The Khronos Group. Opengl shading language. `https://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf`, 2015. Accessed: 2015-02-11.

[Gus05] Stefan Gustavson. Simplex noise demystified, 03 2005.

[Gus12] Stefan Gustavson. Procedural textures in glsl, 2012.

[Hol15] Joachim Holmèr. `http://acegikmo.com/shaderforge/`, 2015. Extension for the Unity ©engine.

[Lew89] J. P. Lewis. Algorithms for solid noise synthesis. *SIGGRAPH Comput. Graph.*, 23(3):263–270, July 1989.

[LLC+10] A. Lagae, S. Lefebvre, R. Cook, T. DeRose, G. Drettakis, D.S. Ebert, J.P. Lewis, K. Perlin, and M. Zwicker. A survey of procedural noise functions. *Computer Graphics Forum*, 29(8):2579–2600, 2010.

[LLDD09] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse Gabor convolution. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)*, 28(3):54:1–54:10, 2009.

[Min86] Marvin Minsky. *The Society of Mind*. Simon & Schuster, Inc., New York, NY, USA, 1986.

[Nor89] Donald A. Norman. Perspectives on the computer revolution, 1989.

[OHH+02] M. Olano, J. C. Hart, W. Heidrich, B. Mark, and K. Perlin. Real-time shading languages. In *Real-time shading languages*, Course 36. SIGGRAPH, 2002.

[Per85] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[Per99] Ken Perlin. Making noise. `http://www.noisemachine.com/talk1/`, 1999. Accessed: 2015-02-10.

[Per02] Ken Perlin. Improving noise. *ACM Trans. Graph.*, 21(3):681–682, July 2002.

[Per05] K. Perlin. Standard for perlin noise, March 15 2005. US Patent 6,867,776.

[PH89] K. Perlin and E. M. Hoffert. Hypertexture. *SIGGRAPH Comput. Graph.*, 23(3):253–262, July 1989.

[Sch15a] Tony Schachman. Adding parabolas. `https://vimeo.com/115785360`, 2015. Accessed: 2015-04-17.

[Sch15b] Tony Schachman. Shadershop. https://github.com/cdglabs/Shadershop, 2015.