CUDA Noise

A CUDA accelerated Procedural Noise Generation Library

Keegan Jones, Shuyu Xia

Contents

1 Introduction

`CUDA_Noise` (working title) is a modular procedural noise generation library written primarily in C++, using CUDA kernels to vastly accelerate the computations performed to generate the noise. The library is "Modular" in that something like a base noise generation module can be transformed and modulated by passing it through a serially-linked set of other modules. These modules can perform simple transformations - like normalization, scaling, or arithmetic operations - or more complex operations, such as linear interpolation between two other modules using a third module as the "alpha" value in the linear interpolation algorithms. The modules are more thoroughly documented, with examples and demo images, in the API reference at the end of this document.

1.1 Purpose

Procedural noise generation is commonly used in graphical applications, along with seeing increasing employment in the video game industry for the generation of assets and in-game items. It also has some surprising applications in scientific research, such as being used to generate noisy random distributions of data for testing neural networks on, generating test environments for N-Body simulators, and even generating large datasets of artificial "tissue" to test machine learning and vision on (for breast cancer research, primarily). Procedural noise is an "embrassingly parallel" problem, meaning that a value at point (X,Y) does not depend on the value at (X + 4, Y - 4), or any other position. The values are all remarkably independent, and given that analytical derivatives are known for most base noise functions there is even less possibility of running into interdependency bottlenecks (i.e stencil computations and finite differencing schemes). Generating large datasets on the CPU, then, is silly and unecessary. As will be shown later, even SIMD programming is unable to match GPU speeds, and threading on the CPU has a number of issues tied to it, especially in terms of dev time and accomodating a broad range of hardware. Thus, the purpose of this project was to demonstrably show that CUDA and GPGPU computational methods would be well-applied to the challenge of noise generation.

1.2 Existing solutions

There are a number of existing noise generation libraries available, but they all are CPU-based. There are some GPU-based noise algorithms, but there are primarily run as elements of shader programs and as such are limited in scope and functionality because of this. As far as the authors are aware, `CUDA_Noise` is the first library of its type. However, it would be unfair to not recognize the previous work in this field, as much of this libraries code is a synthesis of the best methods from each of these solutions.

1.2.1 LibNoise

Libnoise is one of two commonly-used modular noise generation libraries. LibNoise is practically the archetype upon which many other noise algorithms and libraries build - its fairly old, dating back to 2003, but has proven resilient and capable to this day. However, it is a product of its time. It does not feature the improved Simplex algorithm for noise generation - created by the same man who made the Perlin noise algorithm. It's single-threaded, and uses outdated methods of memory management along with being written more like C99 programs than a modern C++ program. That being said, the algorithms used in most of the modules were very nearly directly transcribed into CUDA kernels for our usage, and much of the architecture and methods used to handle modules was taken from this library as well. Available at References 9.

1.2.2 AccidentalNoise

AccidentalNoise is best seen as a more feature-rich and modern version of LibNoise, as it includes the Simplex noise algorithm, uses slightly updated C++ programming paradigms, and includes a number of more useful module types. Additionally, the base noise generation algorithms from AccidentalNoise provided one tremendously beneficial item for our CUDA code - the ability to remove all but one constant memory lookup table, instead using a faster hashing function. Available at References 10.

1.2.3 FastNoise

Unlike the preceding projects, FastNoise is not modular in nature. It is, however, fast (unsurprising, given the name). Benchmarks show it to be 2-3x faster than Libnoise, mainly due to work by the creator to optimize the base generation algorithms and avoid cache-unfriendly methods that litter projects like LibNoise and AccidentalNoise. The lack of modules means that its fairly sparse in terms of feature set, but many applications of procedural noise don't explicitly need the modular approach. Available at References 11.

1.2.4 FastNoiseSIMD

FastNoiseSIMD is a SIMD adaptation of FastNoise, capable of generating noise at even higher speeds. With threading, it is conceivable that it could compete with CUDA Noise given the CPU's proficiency at complicated calculations and the deep level of optimization on the part of the creator of FastNoiseSIMD. However, SIMD operations have some disadvantages. Interfacing with this library is fairly clumsy, and all datasets must be spatially contiguous and have dimensions that are a multiple of 8. Implementing modules would be very tough for SIMD operations too, and the aligned memory allocations oft-required for fast SIMD work can also be tremendously costly. Additionally, the greatest speedup is going to be seen with the AVX2 instruction set, which is not excessively common in consumer processors quite yet. Available at References 12.

1.2.5 Scape

Scape is not a noise library like the other projects detailed thus far - instead, its an application for generating heightmaps. However, it is written in Cg, which is Nvidia's "computational shader" language. This made it invaluable for thinking of ways to optimize our code, especially the noise generation. Its method of using a Texture LUT makes it extremely fast at generating simple noise datasets, and a large amount of time was dedicated to trying to get this functionality to work for us. See References 3 to access the final released program, along with the article series and research documents released by the creator of this program.

1.2.6 GLSL Noise

Also like Scape, GLSL Noise is not a library but rather a collection of noise generation algorithms written in GLSL, the OpenGL shader language. As such, it is like Scape in that it provides examples of noise generation code written for the GPU. Its extremely limited in scope and depth though, and the algorithms don't provide the ability to "seed" the noise functions, which is vital for fully leveraging the benefits of the psuedo-random nature of procedural noise. Its code provided some of the initial test code for things like our Simplex algorithm, but was eventually made obsolete by the synthesis of better solutions. Source code available at References 4.

1.3 Project Scope

Due to time constraints, a limited scope was set for this project. The primary goal was to prove that noise generation algorithms in CUDA would display a significant benefit in terms of computation speed. This goal was easily met early in the project, so secondary goals were set. These were:

- Implement all modules from LibNoise
- Implement the faster, improved base noise algorithm "Simplex Noise"
- Implement 3D noise generation
- Add the expanded module set from AccidentalNoise

The first goal was met in its entirety, as all modules have been successfully ported from LibNoise to our CUDA code. However, one module currently displays limited functionality - the Turbulence module. This is documented further in the section "Issues". The second goal was met as well, with Simplex noise being implemented as a selectable variant of noise to use in our fractal generator objects. Simplex noise seems to be 1.6-2x faster, depending on the use case. It has proven especially useful for the DecarpientierSwiss generator, which uses the analytical derivative of the noise function at a point to great effect. The implementation of 3D noise highlighted problems in our code, and cannot be said to be entirely functional at this time. Baseline functionality for generating 3D noise data was demonstrated, but no attempts were made to implement 3D-capable modules as the base 3D noise tests clearly demonstrated the need for improvement. The last goal was not met at all - an examination of AccidentalNoise did not turn up any modules that seemed crucial, and priority was given to the other goals instead.

2 Noise Algorithms

This section details how the noise algorithms work, from their general algorithmic outline to the specifics of their CUDA implementation, along with subsections dedicated to the various revisions/drafts of these noise algorithms that were used to find the optimal CUDA code.

2.1 Perlin (Gradient) Noise

Perlin noise is one of the most common noise generation functions in use, and of the pair of noise functions in our library it is the more intuitive one. First, the base algorithm will be outline. Then, the various methods this library attempted to implement this algorithm will be reviewed.

2.1.1 Base Algorithm

The algorithm for perlin noise is fairly simple, and goes like this:

1. Given input floating-point input coordinates, split these into integer and fractional components where the fractional component is everything after the decimal point.
2. Generate a psuedo-random gradient vector at each vertex of a unit square (or cube) made up by the integral components.
3. Get the distance vectors from the vertices of the square/cube to the fractional components, somewhere within this unit square.
4. Take the dot product of the distance and gradient vectors at each vertex, giving us our "influence" values.
5. Use the influence values to linearly interpolate (along with using an easing curve to push values to integral results) between the gradient vectors, getting our final result.

A more comprehensive overview of Perlin noise is included as a reference at the end of this document, found at . A link is also included to the original SIGGRAPH document published by Ken Perlin himself on "Improved Perlin Noise". The goal is not to give an in-depth explanation of the algorithm in this document, merely lay some groundwork to justify the methods used to implement this algorithm in CUDA.

2.1.2 First implementation: Baseline

The first functional implementation of Perlin noise in our CUDA code was not heavily modified at all from the dozens of nearly identical implementations that can be found on the web. At this time, the goal was to identify possible issues with the CPU-based algorithm so as to better understand what should be improved. Initial tests displayed the following characteristics:

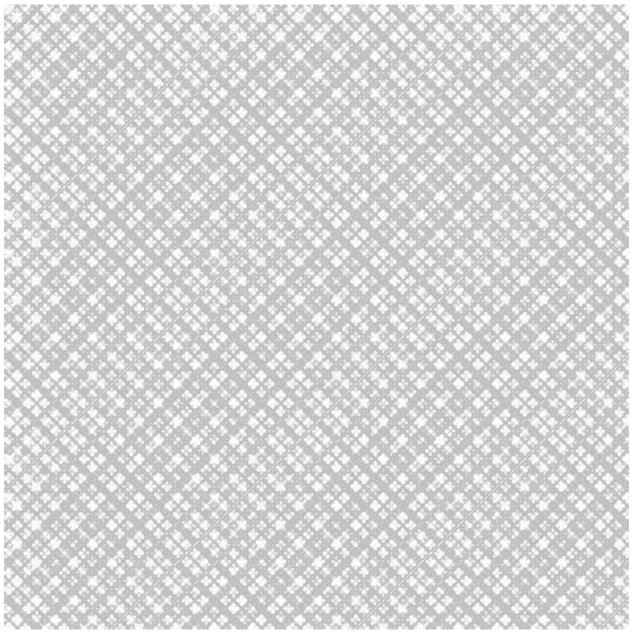
- Register pressure was very high, restricting the grid size immensely

- Due to the small size of the constant memory look-up table, it was stored in the cache
- The baseline ease curve resulted in visible repeating artifacts in the output image

2.1.3 Second implementation: Texture-Memory Look-Up Tables

The second iteration of the Perlin noise algorithm in our library aimed at addressing the first two issues. In Scape's "Procedural Basics" article, a method is discussed for generating Perlin noise that can greatly reduce the computational complexity, by reducing a large chunk of the work required into two texture lookups. Texture memory in CUDA is also quite fast, thread-local, and is cached fairly intelligently by the device while in use. Additionally, this texture can be generated for each module at runtime, as needed, allowing it to be seeded and shuffled slightly so that each instance of a module won't be using the same texture objects. Initial implementations of this algorithm displayed bizarre results, however, and weren't making noise-like results at all: It turns out that texture reads in CUDA had some

Figure 1: This is not procedural noise, but it is noise!



rather odd characteristics, and worse downsides. Additionally, the lack of mathematical operators for vector types in CUDA proved especially painful here, as it required that all calculations on vector types be done component-by-component, creating several errors. The primary issue causing this bug, though, was how we read from the texture:

```
float4 perm = tex2D<float4>(perm_tex, i.x / 256, i.y / 256);
```

The main issue here is that our texture is stored as 8-bit data, meaning each element is a 32-bit RGBA pixel. We should be reading a `uchar4`, and attempting to read and interpret this data as a float4 broke everything and meant that the output image (somehow) became an amalagation of the two lookup texture objects. However, it was mentioned in the previous section that register pressure was a rather large issue. The fix for this issue increased register pressure, back to nearly what it was with the original algorithm. The second issue was trying to divide the `int2`, `i`, by 256 - this was incorrect, as we should modulo it with 256 to keep it in the bounds of our lookup texture. Additionally, as part of this same issue, CUDA wants to retrieve values from the center of the pixel, and will sometimes return a value that is linearly interpolated from adjacent pixels if we don't read from the center. We can read from the center by simply offsetting by 0.50 in the positive XY directions, so this was done in the next revision:

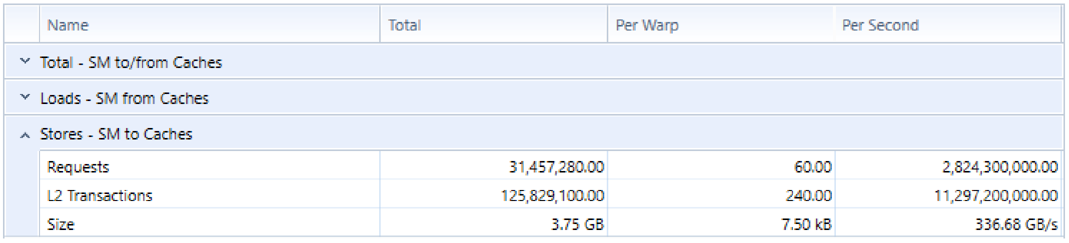
```
uchar4 tmp = tex2D<uchar4>(perm_tex, i.x % 256 + 0.50f, i.y % 256 + 0.50f);
float4 perm = make_float4(tmp.x + seed, tmp.y + seed,
                                        tmp.z + seed, tmp.w + seed);
```

Creating a new `uchar4`, only to use it as a temporary container for conversion to a `float4`, is incredibly wasteful. It gets worse, too, as we actually need to do this twice more when we try to use our other lookup texture. To make it all even worse, this new method ended up far slower than the original method. How? The register pressure caused the grid size to have to be shrunk dramatically, so while the individual threads were much faster, the work as a whole was much less efficient due to the number of grids required. Utilization values in the CUDA benchmark tool dropped as low as 12.5%. This method wasn't working, so the original method was restored.

2.1.4 Third implementation: Libnoise Perlin, Easing Curve Selection

LibNoise has a parameter in it's noise generation modules called "noise_quality". This parameter chooses the type of easing curve to use, from the simplest (linear interpolation), to a 3rd or 5th-degree polynomial easing curve. Not only did the higher quality curves reduce the grid-like artifacting visible in the output considerably, the choice of quality levels meant that (hopefully) quality could be sacrificed for speed. However, the daemon that is register pressure again rose from the grave: the switch statement used to decide on an easing curve method, combined with the local variables required for this, resulted in a large amount of local memory traffic:

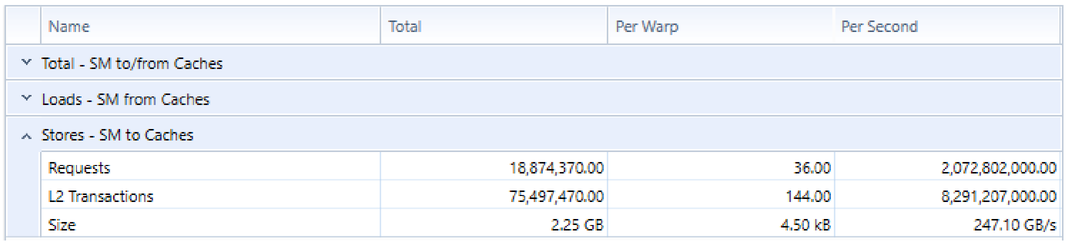
Figure 2: Local memory statistics for Perlin noise kernel without described modifications



Name	Total	Per Warp	Per Second
Total - SM to/from Caches			
Loads - SM from Caches			
Stores - SM to Caches			
Requests	31,457,280.00	60.00	2,824,300,000.00
L2 Transactions	125,829,100.00	240.00	11,297,200,000.00
Size	3.75 GB	7.50 kB	336.68 GB/s

Removing this switch statement and instead going purely with the 5th-degree polynomial resulted in a tremendous reduction in memory traffic (-1.75Gb for a 512x512 test run):

Figure 3: Local memory statistics for Perlin noise kernel with described modifications



Name	Total	Per Warp	Per Second
Total - SM to/from Caches			
Loads - SM from Caches			
Stores - SM to Caches			
Requests	18,874,370.00	36.00	2,072,802,000.00
L2 Transactions	75,497,470.00	144.00	8,291,207,000.00
Size	2.25 GB	4.50 kB	247.10 GB/s

Despite the 5th-degree polynomial easing curve being relatively complex, there was no major loss of speed. Based on detailed profiling data, it looks like the compiler was able to fuse much of this into `__fmadd` operations, reducing the number of required operations per easing curve application considerably. The difference in execution time between the two versions of our code was negligible, and could be easily dismissed as statistically insignificant.

2.1.5 Fourth implementation: Improved Hash method, ”volatile” trick

While browsing the source code of AccidentalNoise, it was noted that AccidentalNoise did not use a permutation table at all. The permutation table is used repeatedly in most Perlin noise algorithms, as coupling it wish a hash function allows for the generation of psuedo-random gradient vectors. However, this hashing is primarily performed during the generation of the permutation table. This action is performed on the CPU, where the table is generated and uniquely shuffled for each module based on the input seed number passed to the module's constructor. In the constructor, the module also takes care of transferring this table to the CPU. Alternatively, one can choose to save a permutation table to constant memory on the GPU, and use the ”seed” number passed to a kernel to still get random values. Contant memory is slow however, and even though it was discovered that it could reside in the cache during the tests of the initial implementation, cache misses are still remarkably expensive. And the cache hit rate was in the low 25% range, consistently. The hope was that implementing the hashing function itself on the GPU in-place of a memory lookup could be used to increase speed, remove the requirement for a device-to-host memory transfer upon initilization of any module, and simplify the code in several places.

However, AccidentalNoise still requires one LUT. This LUT can be made into a short4 datatype though, and then easily integrates with the vector math operators provided by cutil_math.cuh. The hashing method looks like this:

```
// Constants used by hashing algorithm: chosen to cause "avalanching"
__device__ __constant__ uint FNV_32_PRIME = 0x01000193;
__device__ __constant__ uint FNV_32_INIT = 2166136261;
__device__ __constant__ uint FNV_MASK_8 = (1 << 8) - 1;
// Primary hashing algorithm.
inline __device__ uint fnv_32_a_buf(const void* buf, const uint len) {
        uint hval = FNV_32_INIT;
        uint *bp = (uint*)buf;
        uint *be = bp + len;
        while (bp < be) {
                hval ^= (*bp++);
                hval *= FNV_32_PRIME;
        }
        return hval;
}
```

This hashing algorithm performed quite well in initial tests, and there was a 20-30% increase in speed across the board. The cache hit rate increased significantly to 60%, and the ability to use the "short" datatype in a few key locations stopped register pressure from climbing even higher.

2.1.6 Fourth implementation: Improved Hash method, "volatile" trick, contd.

With the improved hashing method addressed, the `volatile` trick will now be mentioned. Apparently, the nvcc compiler likes to put everything into the registers, and hasn't displayed itself as being terribly competent at figuring out which variables can be stored in global memory. It seems to optimize purely for speed, but in doing so actually shoots itself in the foot. The `volatile` qualifier can help with this, however. This states that a certain variable can not be aliased (or have its calculation inlined repeatedly), so it can't be stored in the registers. This does decrease speed, of course, so the usage of the `volatile` qualifier becomes a game of experimentation and fine-tuning. A consistent pattern for the best location to use the `volatile` qualifier was not identified, but its application to a few variables in the base noise generation algorithms stopped CUDA from throwing errors about failed kernel launches.

Unfortunately, that is all there is to say about that topic. Even NVidia employees, in forum threads on this topic, expressed that they didn't understand entirely why this was happening, and a large host of experiments by a rather dedicated forum user were also unable to turn up any consistent usage situations. At the least, the takeaway is that experimenting with the `volatile` qualifier is a worthy pursuit if one is dealing with register pressure issues.

2.2 Simplex Noise algorithm

The Simplex noise algorithm is the product of the creator of the Perlin noise algorithm, but tends to work a little bit better - its a touch faster, has a better analytical derivative (less difficult to implement), and scales very easily to higher dimensions (as high as 6D). However, it uses some advanced math - certainly mostly beyond the author. An attempt was made at explaining the basis of the algorithm, but the article "Simplex Noise Demystified" does a much better job at explaining Simplex Noise, and is written by the author of what has become the de-factor reference/standard implementation of Simplex Noise. This article is linked in the references, as 7.

2.2.1 Base algorithm

The algorithm defining Simplex noise is quite difficult to grasp, but ends up being simpler computationally. The algorithm can be outlined as such:

1. First, choose the appropriate simplex grid for our current dimension. The simplex grid is is tesselated series of shapes, and the shape is chosen as the simplest and most compact shape that can fill the entire space. In 2D, this is simple: a triangle.
2. Skew input coordinates to align more precisely with our simplex grid, separating them into fractional and integral components much like we did for Perlin noise.
3. Generate psuedo-random gradient vectors at each of the three corners of our simplex grid (in 2D), and find the vector distances between the corners and the point inside the simplex cell.
4. Perform the dot product between each gradient and distance vector, getting our values at each corner.
5. As the simplex noise algorithm already radially attenuates, there is no need to linearly interpolate between each value: instead, we can just sum our contributions from each corner. This is where much of the speed increase comes from.

The Simplex Noise algorithm parallels Perlin noise in many ways, which is hardly surprising given that both algorithms were created by the same individual. The Simplex algorithm has a number of advantages though, the greatest of which is the lack of linear interpolation during the final steps of the function. This saves a tremendous amount of computation, and can result in Simplex noise being easily 2-3x faster than naive Perlin implementations. Additionally, the integrated spherical attenuation present when we get values at each simplex grid point also removes the directional artifacting that can haunt Perlin noise - the very artifacting that requires us to use more expensive ease curve and interpolation functions in the first place.

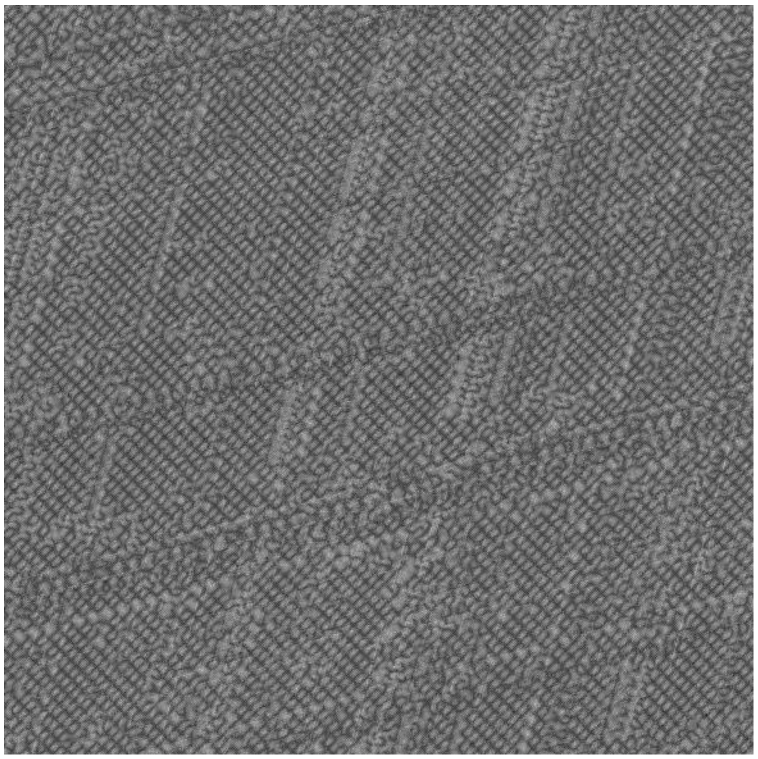
2.2.2 First implementation: GLSL-based Simplex noise

The first attempts at implementing Simplex noise were not done until the Perlin noise algorithms were well developed, and it immediately proved itself as the more difficult of the two. The GLSL implementation was chosen because it seemed likely to keep register pressure low, but could not be ”seeded”. This meant that querying the function at point (x,y,z) would always yield the same result - mostly defeating the purpose of psuedo-random noise, which is to get the same result at the same position WITH the same seed number. The nail in the coffin for the GLSL algorithm was its speed - nearly 2-3x slower than the Perlin noise algorithm, when the inverse should be true! Very little was done to change this algorithm from its implementation available on Github however, so it is left to the reader to seek that out if they wish to see the source code for this algorithm (and, its buried deep in our library's commit history). This can be found at References 4.

2.2.3 Second implementation: Texture-memory lookups

Given that the GLSL algorithm wasn't working, we sought out a solution based on the AccidentalNoise implementation of Simplex Noise (as LibNoise was released before Simplex Noise was debuted at SIGGRAPH). We made a slight change however, choosing to not store the permutation table in constant memory but rather to build this on the CPU. The permutatio table is small, and should've easily fit in the cache. Things seemed fast, but we quickly noticed glaring artifacting in our output images. The artifacts actually looked like the simplex grid in 2D, a number of tiled rhombus shapes:

Figure 4: Note the regular diamond pattern, especially visible where the pattern between cells fails to align



This was clearly a large problem, rendering any sort of output entirely unusable. The indexing of the data was checked numerous times, and a variety of changes/fixes were attempted, but nothing was able to remove this interference. It became one of the ultimate "Heisenbug's" we encountered, as it both existed - in our output - and did not - at the same time (its non-existence due to its failure to respond to all manners of code changes). Ultimately, we abandoned the idea of using the texture lookup object, as it simply could not be fixed and was becoming too much of a time sink with too little reward.

2.2.4 Third implementation: "Reference" implementation

The third, and most successful, implementation was a "Reference" implementation based on the work of Stefan Gustavson, which can be found at References 8. This code is extremely commonly used and can be found almost anywhere Simplex Noise is employed, at its proven to be the most robust of the easily found Simplex Noise implementations on the web. It also includes the analytical derivative, which is an "optional" parameter only calculated when a supplied parameter (pointer to output of derivative segment) is not `nullptr`. This is quite useful in and of itself, as the derivative isn't commonly needed for most of our baseline fractal noise generators, and the calculation of the derivative is relatively costly in the scope of the rest of the function. This code worked quite well for us after some light work to get it functional in CUDA, and formed the first truly "stable" version of Simplex noise we used in our project.

2.2.5 Fourth implementation: Improved Hashing, "volatile" trick

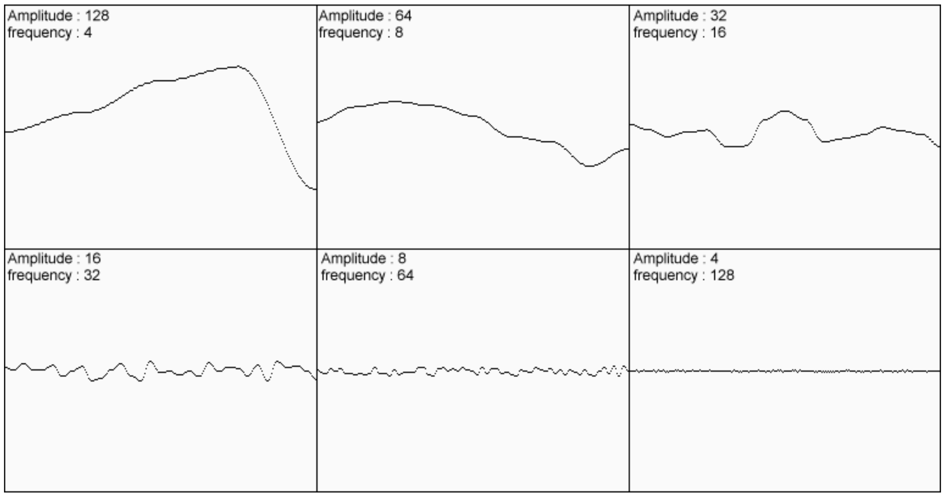
Shortly after the Simplex noise was stabilized, we returned to optimize the Perlin noise algorithm by performing the work that created our fourth implementation of that algorithm. The Simplex Noise algorithm also uses hashing methods though, and was having issues with register pressure as well. We then applied the improved hashing algorithm from AccidentalNoise to our Simplex algorithm as well, and further borrowed from AccidentalNoise by unifying our lookup-tables. This now meant that our Perlin and Simplex algorithms both shared a single lookup table, instead of requiring two lookup tables. Additionally, we were able to leverage the benefits of Stefan Gustavson's "reference" implementation (namely, the analytical derivative) and the AccidentalNoise implementation with a bit of work to mix the best of both implementations. This finalized implementation is the one included in the library currently, and finally gave us the expected speed increase (2-3x over Perlin) that we did not see in earlier implementations.

Curiously, the `volatile` qualifier/trick did not really do anything for our Simplex algorithm. Register pressure did not greatly decrease regardless of where (or how often) we used the `volatile` qualifier. Anytime there was a noticeable decrease in register pressure, there was an undesirable increase in runtime, so the `volatile` qualifier ended up being unnecessary for our Simplex algorithm in the end.

3 Fractal Algorithms

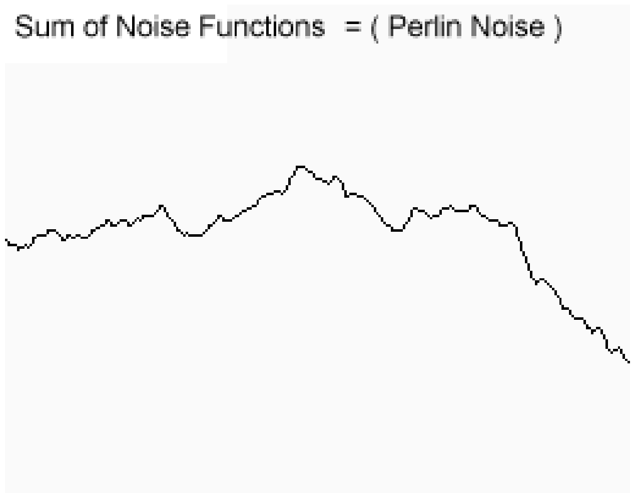
Pure Perlin or Simplex noise by itself is quite boring. All of the generator modules in our library use some sort of fractal iteration system, and this is how nearly every major noise library approaches the issue as well. There simply isn't enough granularity in the output or in our ability to modify the output if we were to simply use pure noise. This is especially apparent in the scale of features, as they tend to be uniform with raw noise. Instead, we use a set of iterative algorithms that increase detail, allow for more customization, and allow for detail at a number of different scales (from mountains to pebbles, so to speak). The core aspect of these algorithms is there use of "octaves", a concept best demonstrated with the following figure:

Figure 5: We generate raw noise data at different frequencies, increasing the frequency for higher "octaves"



As shown in Figure 5, the various output images shown vary greatly in scale and detail. The image with the greatest amplitude has the least detail, and the image with the lowest amplitude has the most detail. Individually, these images are not particularly useful. However, by summing them together we could get better result. However, we want to weight this sum slightly - so that the base image (top-right) sets the overall contour of the output, and the following images only affect this partially. Each of our fractal algorithms has a "persistence" parameter that does just this - before adding the result of the noise function at a certain octave to the final result, we multiply it by the "persistence" value. This value is best kept in the 0.2-0.9 range, and values above 1.0 will produce bizarre results. To complement this weighting, we also have a "lacunarity" parameter. The "lacunarity" parameter helps scale octaves to increase detail by increasing their frequency. A lacunarity value of 2.0, for example, means that each successive octave will have a frequency that is twice the frequency of the preceding octave. Figure 6 demonstrates what we can expect as a result if we applied these ideas to Figure 5:

Figure 6: By using a weighted sum of scaled noise across several octaves, we get vastly improved results



Lastly, what follows is an example of the baseline fractal generator - the "FBM" module. This stands for "fractional Brownian Motion", and is well covered by References 2. Every other fractal algorithm that will be covered is merely an extension of this algorithm.

```
__device__ float FBM2d(float2 point, const float freq, const float lacun,
const float persist, const int init_seed, const int octaves) {
        float amplitude = 1.0f;
        // Scale point by freq
        point.x = point.x * freq;
        point.y = point.y * freq;

        // Accumulate result from each octave into this variable.
        float result = 0.0f;
        for (size_t i = 0; i < octaves; ++i) {
                int seed = (init_seed + i) & 0xffffffff;
                float n = perlin2d(point.x, point.y, seed, nullptr);
                result += n * amplitude;
                // Increases frequency of successive octaves.
                point.x *= lacun;
                point.y *= lacun;
                // Decreases amplitude of successive octaves.
                amplitude *= persist;
        }

        return result;
}
```

3.1 Billow

Billow noise is a very simple modification of FBM noise: we take the absolute value of the noise at each point x,y and add this to the result. This looks like:

```
1  __device__ float Billow2d(float2 point, const float freq, const float lacun,
2  const float persist, const int init_seed, const int octaves) {
3          float amplitude = 1.0f;
4          // Scale point by freq
5          point.x = point.x * freq;
6          point.y = point.y * freq;
7  
8          // Accumulate result from each octave into this variable.
9          float result = 0.0f;
10         for (size_t i = 0; i < octaves; ++i) {
11                 int seed = (init_seed + i) & 0xffffffff;
12                 float n = fabsf(perlin2d(point.x, point.y, seed, nullptr));
13                 result += n * amplitude;
14                 // Increases frequency of successive octaves.
15                 point.x *= lacun;
16                 point.y *= lacun;
17                 // Decreases amplitude of successive octaves.
18                 amplitude *= persist;
19         }
20 
21         return result;
22 }
```

The result is "billowy" because no negative values exist. If used to generate terrain, it will tend to create rolling hills.

3.2 RidgedMulti

Ridged Multi noise is nearly the same as Billow noise - instead of just taking the absolute value of the perlin noise function though, we subtract the value from 1.0 like so. We can just modify line 12 from the Billow example to display this.

```
float n = 1.0f - fabsf(perlin2d(point.x, point.y, seed, nullptr));
```

3.3 DecarpientierSwiss

Previously, one may have noted that the last argument in our calls to ”perlin2d” were simply ”nullptr”. As was briefly mentioned in the last section, this parameter is used for calculating the derivative of the noise at the sampled point. This operation is fairly costly, however, so we supply nullptr to avoid calculating the derivative to save some computational time. DecarpientierSwiss noise, however, uses this derivative to great effect. The algorithm is by the creator of Scape, and a link to the web article from which this algorithm was taken can be found at Ref 2.

```
__device__ float dswiss_perlin(float2 p, const float freq, const float lacun
const float persist, const int init_seed, const int octaves) {
        float amplitude = 1.0f;
        // Scale point by freq
        float2 point;
        point = p * freq;
        // Affects amount we warp the point's position, by scaling power
        // of the derivative sum at the point.
        float warp = 0.01f;
        // accumulate final result into this
        float result = 0.0f;
        // We will accumulate the derivatives into this variable.
        float dx_sum = 0.0f, dy_sum = 0.0f;
        for (size_t i = 0; i < octaves; ++i) {
                int seed = (init_seed + i) & 0xffffffff;
                // Must pass pointer to derivative object: declare
                // before calling perlin2d
                float2 dx_dy;
                // Call perlin2d with our derivative, passed as
                // a ref/pointer
                float n = perlin2d(point.x, point.y, seed, &dx_dy);
                // Base result is just RidgedMulti
                result += (1.0f - fabsf(n)) * amplitude;
                // Increment dx and dy sums by weighted derivative,
                // also multiplying by negated  value for this octave
                // and position
                dx_sum += amplitude * dx_dy.x * -n;
                dy_sum += amplitude * dx_dy.y * -n;
                // Scales frequency of succeeding octaves
                point.x *= lacun;
                point.y *= lacun;
                // Move point based on derivative and warp strength.
                point.x += (warp * dx_sum);
                point.y += (warp * dy_sum);
                // Additional step: saturate result (clamp to 0-1.0 range),
                // multiply by persist. Decreases power of results
                // when result is small.
                amplitude *= persist * __saturatef(result);
        }
        return result;
}
```

The Decarpientier-Swiss algorithm is rather complex, so an attempt to explain how it works is in order. More thorough explanation of this can be found at References 2.

- ”dx_sum” and ”dy_sum” are weighted similarly to ”sum”, but the result of multiplying by the gradient (dx_dy) and negated ”n” is that features on ridges are elongated and exaggerated.
- To make valleys smoother, we multiply by the intermediate value result in-between octaves, making detail rapidly decrease in flat/valley regions.
- Effectively, we use the noise gradient to increase detail where the gradient is large and decrease detail where the gradient is small.

The goal of this algorithm is to remove the unnatural and artifical look that a lot of generated terrain can have, as everything looks ”fractal”-ized, with detail where there shouldn’t be detail (at least, not if erosion exists). Compare and contrast Figure 7 and Figure 8, showing terrain generated by the RidgedMulti algorithm and the DecarpientierSwiss algorithm, respectively.

The raw output from this algorithm can also vary wildly based on the magnitude of the warp parameter. High warp values will produce output that looks almost cellular in nature, and low warp values produce something that looks a bit more ”tame”. In both cases, it can be clearly seen that algorithm succesfully increases detail and feature density in regions that are ”ridges”, and sharply decreases them in regions that are smoother. Figure 9 shows output generated with a warp value of 0.15, and Figure 10 shows output generated with a warp value of 0.03. At some point, our Decarpientier-Swiss method should be expanded to include this as a user parameter, but for now it was left at the low value to avoid surprising a user with bizarre results.

Figure 7: RidgedMulti terrain: note the detail even in the valleys

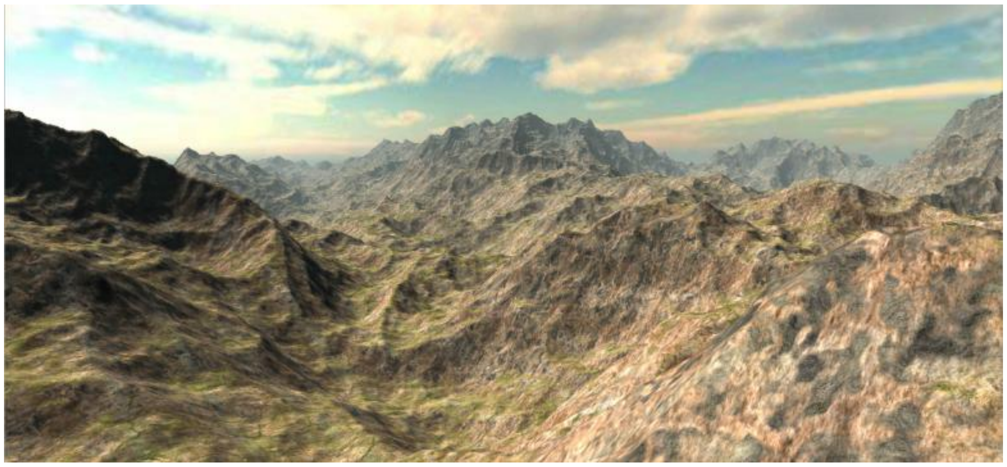


Figure 8: DecarpientierSwiss Terrain: note the smooth valleys, and craggy peaks

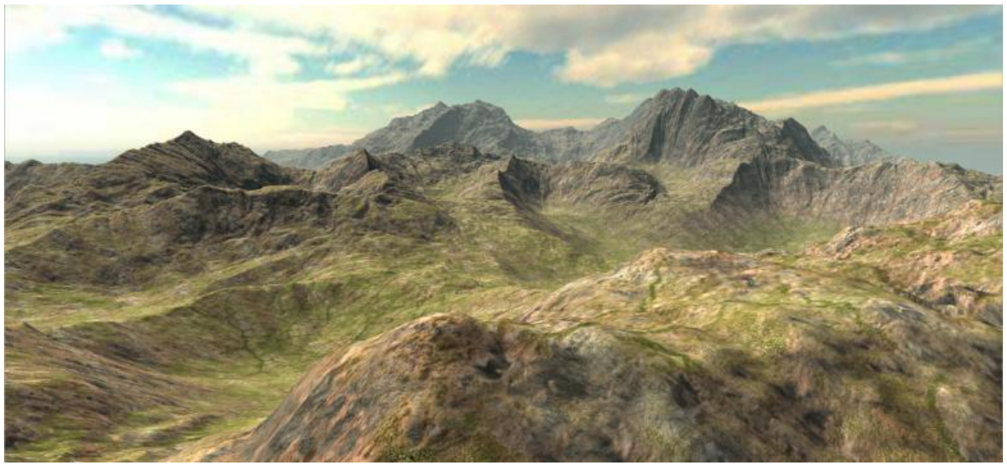


Figure 9: High warp values tend to produce structures that appear cellular

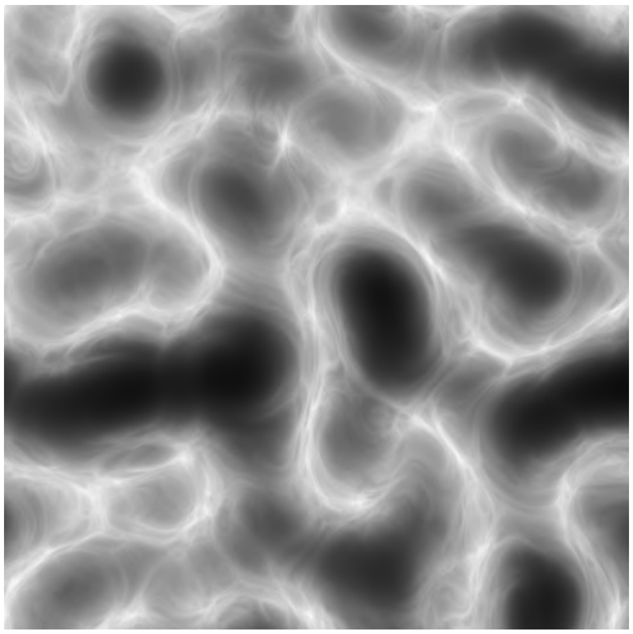
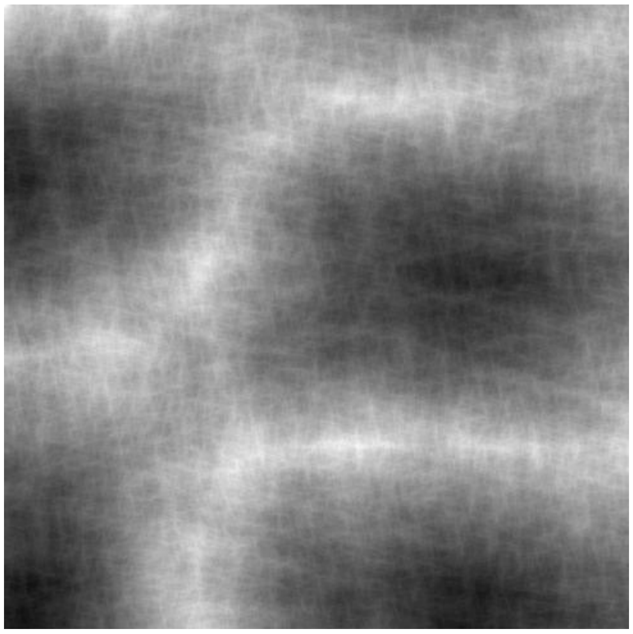


Figure 10: Low warp values look like an enhanced version of RidgedMulti output



4 Challenges

This section will address the key challenges we encountered when working on our project, and how we attempted to either fix work around these problems or fix/remove them outright.

4.1 Surface and Texture Objects

If one looks back at the first few ”working” versions of the library, they will note that each kernel takes arguments of `cudaSurfaceObject_t`’s as output storage arrays, and `cudaTextureObject_t`’s as input variable arrays (to get input from previous modules). This was initially because it was believed the sky-high thread locality of texture and surface object memory would greatly increase speed - a thread with index (x,y) is likely to read pixel (x,y), which means its very likely that the value it needs to read from the `cudaTextureObject_t` input is highly local, and in turn means that the memory location in the `cudaSurfaceObject_t` output that it writes to is also highly local. Initial tests showed cache hit rates in the high 80% range, which verified that there was some benefit to the locality at least.

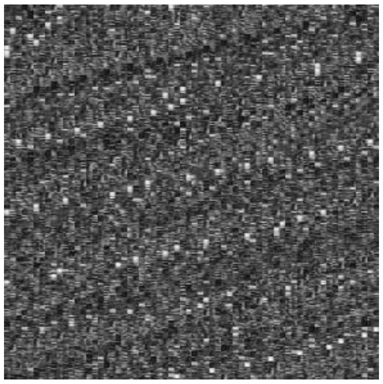
However, `cudaTextureObject_t`’s have some issues, namely that they are read-only and that managing their allocation and eventual deallocation is difficult, not to mention the process of translating a surface object into a texture object. Another advantage commonly exploited with `cudaTextureObject_t`’s and `cudaSurfaceObject_t`’s is how easily they work with graphics API interop processes: its really quite trivial to pass the location of some texture memory to a graphics API like OpenGL, and its even easier to use this texture data as, well, texture data for rendering. But our library was not scoped to work with graphics API’s, and the issues surrounding managing the memory belonging to a large collection of these objects was crippling our ability to create long module chains for testing. Additionally, it could be a bit tricky (and dangerous) to attempt to grab this data for usage by the CPU (e.g for image writing), as sometimes the location of a texture object was in use by CUDA when the CPU wanted to read from it to get a copy (causing errors to abort the program).

Thus, it was decided that we would move to using `cudaManagedMalloc()` with a regular `float` pointer array. Managed memory turned out to be a breeze to work with, and allievated all of our issues nearly on the first compile. Translating the code was a breeze, and there was no longer a need to distinguish betwen a `cudaTextureObject_t` and `cudaSurfaceObject_t` when considering the arguments passed to/from the various methods in our library. To truly seal the deal, there was practically no difference in speed between the two types of memory.

4.2 Turbulence module

In LibNoise, the Turbulence module creates something that can only be described as a ”turbulent” effect on an input module, perturbing the image and generally increasing detail. This is done by slightly translating each sampling point in an image by a very slight amount, with this amount found using either perlin or simplex noise (see Code Sample **??**). This rapidly failed with CUDA, though: its really easy to index out of bounds of our managed-memory array, and things got really tricky really fast because CUDA gladly expanded the array. This confounded the CPU when it tried to access and copy the data back, and caused errors to throw in the memcpy assembly code that's part of the standard C++ library. If this didn't happen, the output became so corrupted that it was (ironically) most like white noise (Figure 11).

Figure 11: The Turbulence module tended to fail rather catastrophically



One of the first attempted solutions was using the modulo `%` operator. This was applied before indexing into the source array, to get a value at a randomized position but ensure that it was still within the bounds of said array. This stopped us from getting values that would index out of the array, except when the values were negative. The solution for dealing with negative values was performing a simple conditional check for values less than 0, and then "wrapping" these values so that subtracted from the max width and height of our array if they happened to be less than zero. Together, these quick fixes allowed us to get turbulence that continued to work even at extremely high powers - compare Figure 12 with Figure 13, which is the before/after of a test heightmap when run through our turbulence module.

Figure 12: Base image fed into Turbulence module

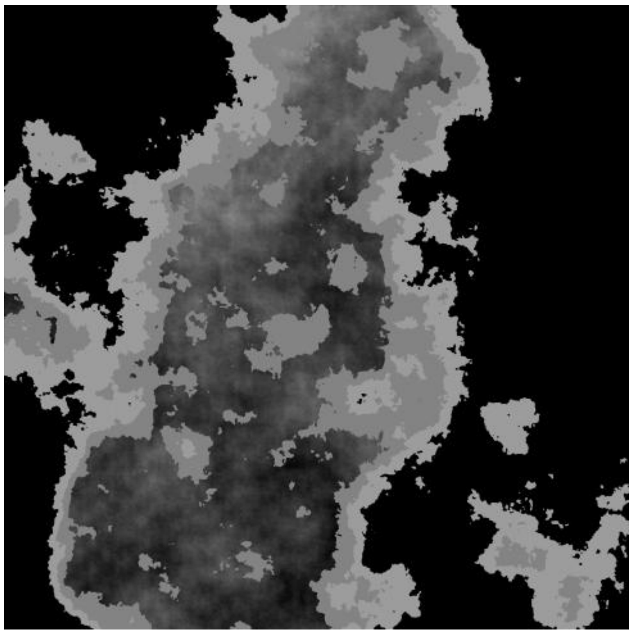
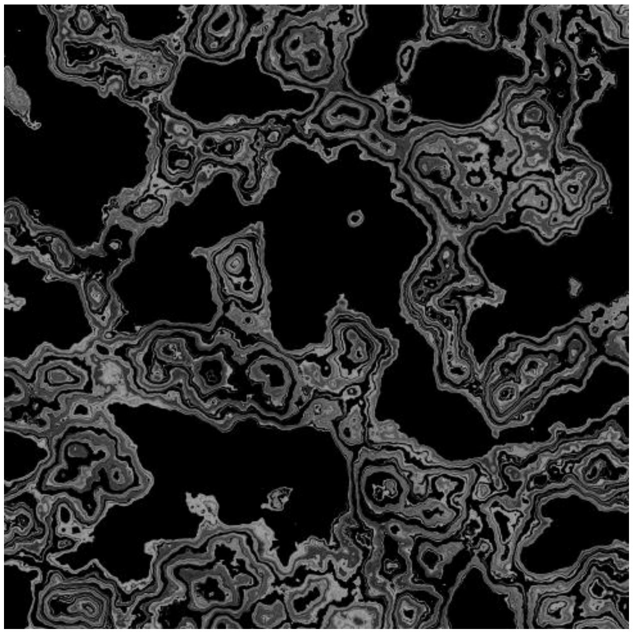


Figure 13: The previous figure run through a Turbulence module, at excessively high power



CUDA Kernel code for our finalized Turbulence module

```
/*
        Turbulence process:

        1. Get current pixel position.
        2. Offset pixel position using perlin/simplex noise.
        3. Before reading from input with new pixel position,
        make sure its in bounds
        4. Read from input with offset position, and use this
        value to set the (i,j) position to this new value
        in output.
*/

__global__ void turb_kernel(float* out, const float* input, const int width,
const int height, const cnoise::noise_t noise_type, const int roughness,
 const int seed, const float strength, const float freq) {
        // Get current pixel.
        const int i = blockDim.x * blockIdx.x + threadIdx.x;
        const int j = blockDim.y * blockIdx.y + threadIdx.y;
        // Return if out of bounds.
        if (i >= width || j >= height) {
                return;
        }
        // Position that will be displaced
        float2 distort;
        if (noise_type == cnoise::noise_t::PERLIN) {
                float n = FBM2d(make_float2(i, j), freq, 2.20f, 0.90f,
                                                        seed, roughness)
                distort.x = n * strength;
                // Resample at slightly different position so that Y
                // isn't translated same dist/dir
                n = FBM2d(make_float2(i + seed, j + seed), freq,
                               2.20f, 0.90f, seed, roughness)
                distort.y = n * strength;
        }
        else {
                float n = FBM2d_Simplex(make_float2(i, j), freq, 2.20f,
                                                0.90f, seed, roughness)
                distort.x = n * strength;
                // Resample at slightly different position so that Y
                // isn't translated same dist/dir
                n = FBM2d_Simplex(make_float2(i + seed, j + seed), freq,
                                        2.20f, 0.90f, seed, roughness)
                distort.y = n * strength;
        }
        // Get offset coordinates by adding distort amt to (i,j).
        int i_offset, j_offset;
        i_offset = i + distort.x;
        j_offset = j + distort.y;
        i_offset %= width;
        j_offset %= height;
        // If offset is negative, subtract it from max width/height
        // to "wrap" around bounds.
        if (i_offset < 0) {
                i_offset = width + i_offset;
        }
        if (j_offset < 0) {
                j_offset = height + j_offset;
        }
        // Use offset position to get value from input, setting value
        // of output at (i, j) to be offset value from input.
        out[(j * width) + i] = input[(j_offset * width) + i_offset];
}
```

5 Future Work

This project has raised a number of possibilities for possible improvements and expansions to the core featureset, which will be quickly reviewed here.

- 3D Noise generation is currently restricted to only being spatially continuous arrays of locations, and offers no way to sample a scattered set of locations (like a point cloud). This is less than optimal, and does not reflect common usage situations for 3D noise in the slightest. Currently, effort is underway to replace the managed-memory `float*` object used as the main datamember with a `Point*` object, also allocated with `cudaManagedMalloc()`. A `Point` is just a struct containing coordinates and a value, allowing the CUDA kernel to act as it needs. This includes reading a position and writing a value with the noise generators, and or just reading the value and performing some sort of basic operation in the case of many of our other modules.

- Graphics API interoperability. `cudaManagedMalloc()` apparently still allows one to get the pointer to device memory associated with the object we allocate for - if we could find a way to get this pointer and then pass it to a graphics API, it would become extremely easy to use the output from this program in graphics applications without requiring costly memory transfers from the GPU to the CPU, followed by copying the same data to a new location that the graphics API knows of. Cutting out that awkward and expensive middle step seems ideal.

- The CUDA runtime API is what was used in this project, which decreases dev time by increasing abstraction. The CUDA driver API is considerably more difficult to use, but allows for more explicit control over resources (like memory) and for the loading of CUDA code as "modules". Given the already modular nature of our program, being able to selectively link together appropriate CUDA resources at runtime based on the noise modules being used would hopefully increase performance, and possibly allow for more efficient resource usage. An investigation of the CUDA driver API's capabilities should be considered.

- Sometimes, one wants to get image output of a module that is in the middle of a long module chain. This currently requires halting all computation on the GPU to do a simple and safe memory copy back to the device, at which point computation can be asynchronously resumed. `cudaManagedMalloc()`'d objects allow for async operations though, which could let us copy data back to the CPU without stopping the GPU for quite some time. This requires a number of careful safety checks, though, along with some more advanced API calls that there was not sufficient time for.

References

[1] Giliam J.P. de Carpentier, Scape 2: Procedural Basics,
`http://www.decarpentier.nl/scape-procedural-basics`

[2] Giliam J.P. de Carpentier, Scape 3: Procedural Extensions,
`http://www.decarpentier.nl/scape-procedural-extensions`

[3] Giliam J.P. de Carpentier, Scape 5: Overview and Downloads,
`http://www.decarpentier.nl/scape`

[4] Stefan Gustavson, webgl-noise,
`https://github.com/ashima/webgl-noise`

[5] Adrian's Soapbox, Understanding Perlin Noise,
`http://flafla2.github.io/2014/08/09/perlinnoise.htm`

[6] Hugo Elias, Perlin Noise,
`http://freespace.virgin.net/hugo.elias/models/mperlin.htm`

[7] Stefan Gustavson, Simplex Noise Demystified,
`http://weber.itn.liu.se/s̃tegu/simplexnoise/simplexnoise.pdf`

[8] Stefan Gustavson, ”sdnoise1234.c”, retrieved from,
`http://weber.itn.liu.se/s̃tegu/simplexnoise/DSOnoises.html`

[9] Jason Bevins, libnoise ”coherent noise-generation library”,
`libnoise.sourceforge.net`

[10] Joshua Tippets, AccidentalNoise,
`http://accidentalnoise.sourceforge.net/`

[11] Jordan Peck, FastNoise,
`https://github.com/Auburns/FastNoise`

[12] Jordan Peck, FastNoiseSIMD,
`https://github.com/Auburns/FastNoiseSIMD`